UNIVERSITÀ DEGLI STUDI DI PADOVA
FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA MAGISTRALE IN INFORMATICA

Tesi di Laurea

Analisi e sviluppo di un'applicazione distribuita per l'elaborazione dei dati nell'ambito del mondo del fashion

Relatore:

Prof. Sergio Congiu

Laureando:

Riccardo Lorenzon

ANNO ACCADEMICO 2010-2011

Indice

Capitolo 1

Introduzione

La tesi è stata svolta nell'azienda H-Farm di Treviso, primo incubatore di start-up a capitale privato in Italia. H-Farm raccoglie al proprio interno decine di aziende fornendo supporto dalle risorse finanziarie a tutta una serie di servizi e logistiche per permettere a queste una rapida crescita nel mercato. Attualmente l'azienda è presente con una propria sede in Italia, Regno Unito, Usa e India, aree di mercato molto diverse tra loro per cultura e società e quindi adatte per una logica commerciale globalizzata.
La società interna di H-Farm in cui è stato svolto l'elaborato è H-Umus, essa si occupa dello sviluppo e della progettazione di sistemi personalizzati per le aziende operanti nel mondo del fashion a partire dalla programmazione back-end fino al client-side. I punti forti dell'azienda risiedono nell'ecletticità, nella grande esperienza nel settore ed inoltre nello sviluppo di client su piattaforme quali Apple IPhone e IPad.
Oltre ai progetti esterni è in fase di sviluppo un progetto interno relativo alla creazione di un framework per applicazioni che sia facilmente personalizzabile per ogni cliente sfruttando il concetto di configurabilità. Tuttavia non è stata ancora fissata un'idea sulla base di dati adatta all'obiettivo di omogeneità proposto ed inoltre, per quanto riguarda il lato server, sono da analizzare delle strutture che possano essere utilizzate per il processing di grosse quantità di dati in un tempo relativamente breve. In questo ambito si è svolto il lavoro di tesi, ovvero un progetto di ricerca e sviluppo di un'applicazione che fosse globale ma adattabile alle esigenze di ogni cliente attuale dell'azienda, che fosse modulare e quindi riutilizzabile, configurabile con semplicità ed infine pensata e realizzata per una struttura di calcolo centralizzato(Apache) e distribuito(Google App Engine).
La tesi può essere descritta procedendo per step successivi.
Nel primo step il compito è stato quello di produrre un database dinamico, ovvero utilizzabile per qualsiasi realtà attuale dell'azienda attraverso una costruzione dinamica degli attributi relativi ad ogni singola entità. Nello step successivo è stata implementata una web application che, a partire dal catalogo del cliente in formato xls o csv permette di generare una nuova applicazione dedicata.
In particolare l'applicazione principale deve permettere di configurare il metodo di migrazione dei dati dallo spreadsheet al database a seguito della sincronizzazione tra il servizio relativo al foglio elettronico e il server contenente il datastore. Si passa quindi da un insieme di dati generico e poco utilizzabile ad un vero e proprio schema strutturato in grado di garantire performance nelle query e nella navigabilità di alto livello. Questa implementazione è stata compiuta utilizzando l'architettura di Django, un potente framework open-source per la programmazione back-end e Apache, web-server anch'esso open-source. Il passo successivo è stato quello di effettuare l'implementazione nell'architettura Django/Google App Engine, dove App Engine consiste in un servizio di Cloud Computing messo a disposizione da Google. Le diversità tra App Engine e Apache in termini di datastore, filesystem e processing dei dati hanno portato a riconsiderare significativamente l'analisi e l'implementazione effettuata. L'ultimo step infine è stato quello di sincronizzare le web app generate con un'applicazione client-side in ambiente iOS e device iPad per la visualizzazione dell'output prodotto dal processing.
Il periodo di lavoro è stato complessivamente da Settembre a Gennaio compresi. Sono state definite delle deadline precise per la presentazione del lavoro svolto al team di sviluppo. Le scadenze sono state fissate rispettivamente a fine settembre, al 12 ottobre, al 3 novembre e al 13 gennaio.

Ogni fase del lavoro verrà descritta nel seguente modo: background sulle conoscenze necessarie, un'analisi sugli strumenti utilizzati, sugli obiettivi e sui requisiti e la descrizione dell'implementazione vera e propria. Infine, come conclusione della descrizione dello step, i risultati e l'avanzamento nel progetto finale.

Capitolo 2

Creazione modello di dati globale

2.1 Background

2.1.1 Formato Csv

CSV (comma-separated list o comma-separated variables) è un formato di file utilizzato per memorizzare dati nei fogli elettronici e nei database e per lo scambio di informazioni strutturate tra le applicazioni.
La diffusione capillare dell'utilizzo di strutture ordinate di immagazzinamento dati ha reso questo formato uno standard per ogni piattaforma.
La memorizzazione e la suddivisione dei dati viene implementata utilizzando il carattere ',' (virgola, da cui il nome comma-separated-values) per separare i valori contenuti in ogni riga. Essendo quindi i dati suddivisi in campi diversi il file di testo assume una formattazione tabulare riconoscibile dai fogli elettronici e dai database. Può essere visto infatti come una singola tabella in cui i record corrispondono alle righe e gli attributi alle colonne.
Secondo lo standard comune il carattere separatore è quindi la virgola, nel caso si consideri un campo di elementi che possono contenere il carattere ',' viene utilizzato il carattere di escape '"'(double quotes). Nel caso un campo preveda dati comprendenti il carattere double quotes viene utilizzato nuovamente il carattere di escape precedente. Le righe sono separate dal carattere '\n'(dall'inglese new line che sta per 'a capo').
Questo tipo di file permette di formattare lo spazio degli elementi di un comune file di testo in maniera tale essi siano visualizzabili come campi di una tabella mediante l'utilizzo di un qualsiasi editor testuale. Proprio per il fatto che possono contenere una sola tabella questi file assumono la denominazione 'flat file'(file piatti).
I linguaggi di programmazione ad alto livello, come python, permettono di utilizzare un vasto insieme di API per interagire con le funzionalità del flat file consentendo quindi di lavorare con una vera e propria base di dati.
Nel progetto è stato scelto di importare i dati utilizzando questo formato in quanto è il formato standard utilizzato per la catalogazione da parte dei clienti ed inoltre è supportato dai servizi di Google Docs.

2.1.2 Formato JSON

JSON (JavaScript Object Notation) è un formato adatto per lo scambio dati in applicazioni client/server che utilizza la notazione letterale di Javascript per la rappresentazione di dati strutturati.
La differenza tra il formato precedente è insita nella semplicità di memorizzazione dei dati: mentre il formato CSV separa gli attributi con un semplice carattere separatore JSON permette di realizzare una vera e propria sintassi e quindi una struttura molto più complessa.
I punti di forza di questo formato sono l'utilizzo di primitive ben supportate dai browser più comuni che ne rendono efficiente la gestione e la totale indipendenza dal linguaggio di programmazione utilizzato legata tuttavia ad un vasto framework presente in tutti i linguaggi ad alto livello.
A differenza del formato csv in json i dati vengono specificati utilizzando tre strutture fondamentali:

- object: ovvero una collezione di coppie <chiave,valore> separate da virgola e racchiuse all'interno di parentesi graffe.
- array: sequenza ordinata di valori omogenei separati da virgola

- value: un singolo valore che può essere una stringa racchiusa tra virgolette, oppure un numero, un oggetto o un array.

Nel corso del progetto questo formato di dati è stato utilizzato in primo luogo in quanto l'analisi ha coinvolto delle implementazioni esistenti di database remoto. Per ottenere il modello relazionale è stato esportato il modello di dati producendo un file json per la rappresentazione testuale.
Viene riportato di seguito un esempio di entità descritta in un file JSON secondo un protocollo aziendale (che verrà illustrato nel capitolo cinque):

```
{"entity" : "ItemStatus",
  "joinModel" : false,
  "attributes" : [
     {"name" : "available",
       "type" : "boolean",
       "optional" : true},
      {"name" : "description",
       "type" : "string",
       "optional" : true },
      {"name" : "server_id",
       "type" : "string",
       "optional" : true},
      {"name" : "itemsizeinfos",
       "type" : "hasMany",
       "optional" : true,
       "destEntity" : "ItemSizeInfo",
       "lookupKey" : "server_id",
       "deleteRule" : "",
       "inverse" : "status"}]},
```

Dallo studio di documenti contenenti codice definito con una sintassi di questo tipo è possibile ricavare lo schema del modello relazionale associato.

2.2 Implementazione

2.2.1 Strumenti utilizzati

Lo sviluppo di questa parte è stato soprattutto teorico, per l'analisi sono state utilizzati comunque gli strumenti di Microsoft Excel e Microsoft Access quali le tabelle Pivot e il supporto per la dichiarazione di query.
La tabella Pivot[1] permette di visualizzare l'insieme dei dati contenuti in una colonna del file excel raggruppandoli per categorie. Dove con categorie si intendono i valori contenuti in un'altra colonna. Questi dati possono essere inseriti attraverso l'indicazione del conteggio o di altri calcoli quali per esempio la media, la percentuale nel totale degli associati a quella specifica categoria e la deviazione standard.

2.2.2 Analisi dei requisiti e obiettivi

Lo scopo del lavoro è quello di creare una base di dati che sia:

- Globale: ogni altra base di dati deve essere mappata in essa con il minimo processing ed il minimo spazio eccedente. Questo per permettere la completa compatibilità con i dati attualmente memorizzati in altre strutture.

- Denormalizzata: nonostante sia la normalizzazione nelle tre forme normali una delle proprietà fondamentali per la facilità di lettura e soprattutto per l'aggiornamento e il controllo dell'integrità del database, essa comporta una sensibile diminuzione delle prestazioni per quanto riguarda l'accesso ai dati stessi.
  Sono stati sviluppati degli studi[2] che dimostrano come il tempo di risposta delle query che implicano join multipli aumenti linearmente.
  Nelle applicazioni web, e, soprattutto, per l'applicazione che verrà sviluppata nel corso di questo progetto, la maggior parte delle operazioni che vengono eseguite sono di lettura. Per questo motivo l'obiettivo è quello di sviluppare il database e portarlo in forma normale unificando successivamente tutte le entità possibili per eliminare i join multipli invocati nelle query più frequenti.
  Essendo inoltre uno degli step successivi il porting in App Engine che, come verrà descritto nella sezione dedicata, utilizza come dbms BigTable, il numero di relazioni tra le tabelle e quello delle tabelle stesse sono parametri negativi per l'efficienza nell'inserimento dei dati essendo il gestore della base di dati non relazionale(la spiegazione è rimandata al capitolo quattro dedicato interamente a GAE).
- Scalabile: l'inserimento deve essere lineare nel numero di righe del catalogo iniziale. Dovendo elaborare fogli di calcolo contenenti decine di migliaia di righe questa è una condizione necessaria per i criteri di efficienza imposti.
- Dinamica: ogni cliente può avere un insieme di attributi relativi ai propri prodotti variabili per numero e tipo. Non è consigliabile quindi memorizzarli in una struttura fissa in quanto questo porterebbe ad un grosso spreco di memoria centrale e secondaria oltre alla naturale difficoltà di gestione del database a livello fisico.

La prima parte del lavoro è consistita nello studio delle realtà relative al mercato commerciale dell'azienda.
Dallo studio sono state individuate gli elementi in comune tra i database, i cataloghi e i metadati forniti ed infine è stato progettato un database globale in grado di permettere il mapping di tutti questi dati in modo efficace ed efficiente nelle proprie tabelle secondo le caratteristiche illustrate.
Le prime analisi hanno considerato dei file csv contenenti decine di migliaia di righe. Successivamente sono stati esaminati degli schemi relazionali già esistenti e dei file json contenenti metadati.
Per ogni cliente attuale viene fornito uno schema relazionale ottenuto dal materiale fornito.
Successivamente vengono analizzati i fattori positivi e negativi e gli aspetti rilevanti nella creazione del progetto finale relativi all'analisi effettuata.

2.2.3 Analisi database esistenti

2.2.3.1 Analisi database esistenti: Armani

Nel caso dei database Armani l'analisi è partita da dei file csv composti da decine di migliaia di righe. Questa proprietà rendeva di fatto impossibile un'analisi standard senza l'utilizzo di strumenti dedicati. Per questo motivo sono stati utilizzate le query di Microsoft Access e le tabelle pivot dei fogli di calcolo di Microsoft Excel.
Le tabelle Pivot in particolare sono state utilizzate per impostare dei raggruppamenti per gli attributi di ogni singola riga dello spreadsheet. In questo modo è stato possibile identificare le dipendenze tra attributi, le relazioni tra le entità e modellare uno schema entità relazione che avesse un buon comportamento nell'inserimento dei dati del file csv.
Gli attributi relativi alle entry del file sono: brand, anno, stagione, collezione, descrizione collezione, segmento, descrizione segmento, categoria, descrizione categoria, prodotto, descrizione prodotto, tipo prodotto, descrizione tipo prodotto, specifiche del prodotto, descrizione specifiche del prodotto, ordinamento merceologico, descrizione ordinamento merceologico, raggruppamento merceologico, descrizione raggruppamento merceologico, modello, tessuto, colore, tipo variante, variante, codice

drop, consegna, descrizione articolo, produttore, descrizione produttore, annullamento.
Già da questa prima collezione di dati si evince come il numero di attributi relativi ad una singola entità possa essere vario e articolato. Assunzione che assume maggior valore se si pensa che tutti gli attributi sono relativi al solo catalogo, mancano infatti informazioni relative all'inserimento di clienti, agenti, ordini e alle immagini relative ai prodotti.
L'elaborazione ha portato allo schema seguente:

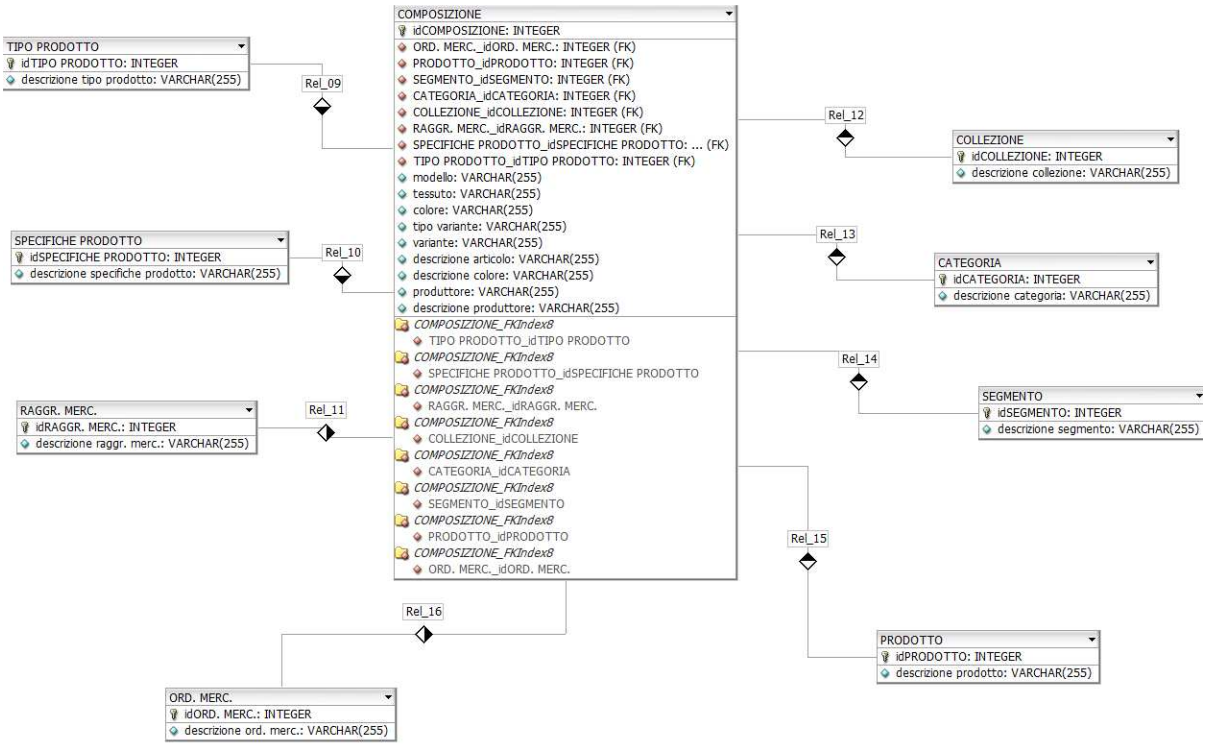

Figura 1: modello relazionale Armani

Dall' analisi è stato possibile derivare che molti attributi sono raggruppati in più classi e dipendono in modo relazionale. Inoltre è stato derivato che la tripla <*modello, tessuto, colore*> rappresenta una chiave primaria per ogni oggetto.
Da questo primo modello è facile capire, inoltre, come sia necessario, per soddisfare i requisiti di omogeneità, conservare un numero di attributi rilevante e difficilmente gestibile.

In questo caso, come negli altri analizzati, è stato derivato uno schema relazionale normalizzato. Proprio per la caratteristica dei modelli presenti ,e la necessità di modificarne la struttura basilare, la ricerca di una soluzione globale è stata focalizzata sulla dinamicità e la velocità di esecuzione delle query di ricerca.
Le basi di dati analizzate vengono utilizzate quindi per lo studio degli attributi e delle entità contenute in esse e per la successiva mappatura dei concetti espressi nel database finale.

2.2.3.2 Analisi database esistenti: Comete

Per l'azienda Comete l'analisi è partita da un file json contenente la descrizione strutturale del database. Dallo studio dei metadati è stato derivato uno schema relazionale completo:

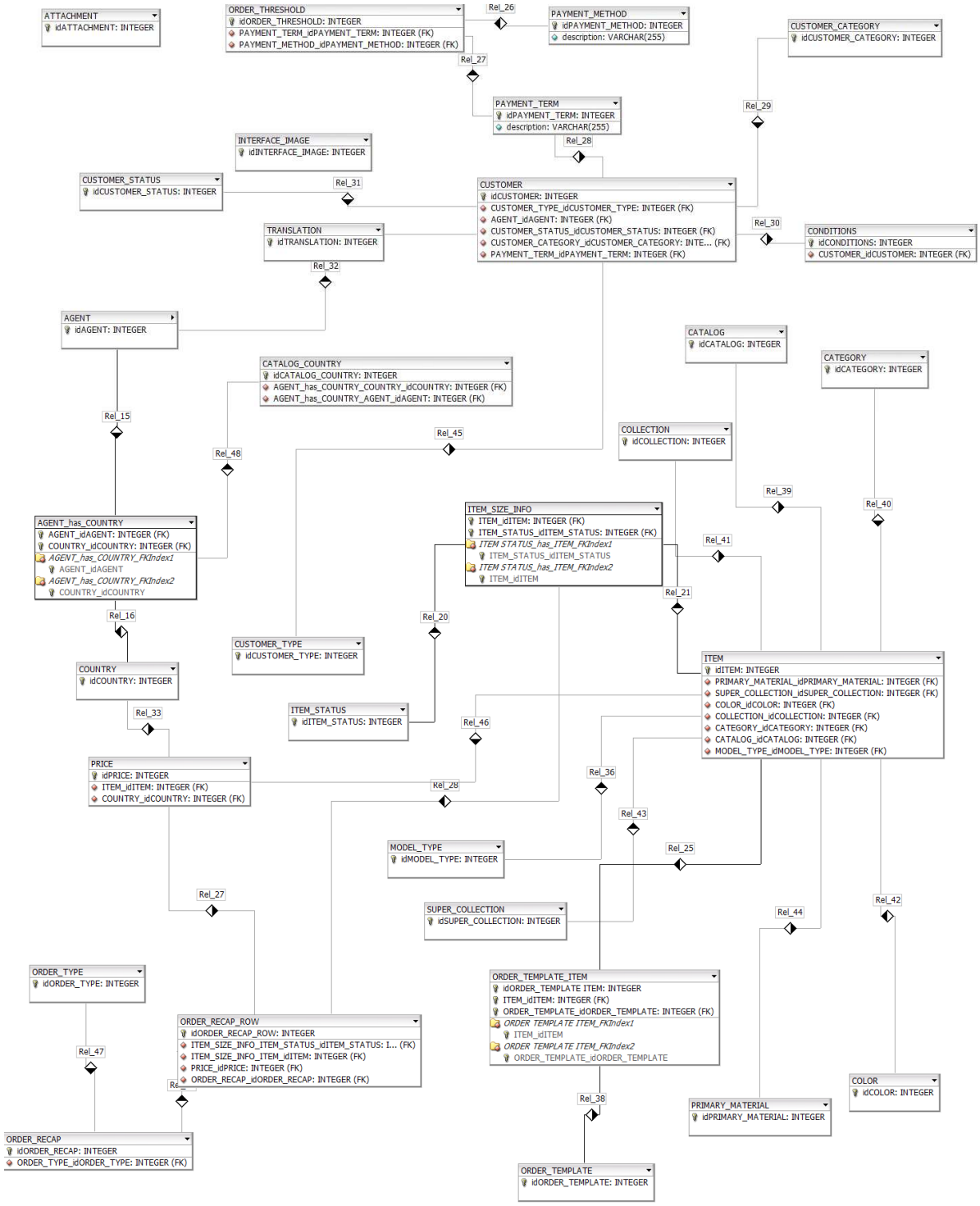

Figura 2: modello relazionale Comete(gli attributi sono stati omessi per aumentare la leggibilità)

In questa base di dati, rispetto alla precedente, sono stati mappati gli aspetti relativi alla parte commerciale: gli ordini di pagamento, il cliente e l'agente di vendita.
Il numero di entità relative al catalogo invece è minore rispetto al modello Armani. Nonostante questo gli attributi relativi sono diversi, compaiono infatti le entità SUPER_COLLECTION, CATEGORY oltre a ITEM_SIZE_INFO.
Per gli obiettivi proposti il modello globale deve dare la possibilità di memorizzare tutte queste informazioni.

Inoltre compaiono le entità ATTACHMENT e TRANSLATION, l'utilità di queste è data dal supporto fornito alla base di dati. La prima consente di associare ad ogni entry del modello un qualsiasi dato multimediale sia esso un'immagine, un video, un file musicale.
Nell'ambito di commercializzazione del prodotto sviluppato la visualizzazione grafica è molto importante e quindi nello schema logico della base di dati globale devono essere previste le entità necessarie alla memorizzazione degli allegati.
TRANSLATION invece è stata creata per risolvere il problema dei dati provenienti da aziende dislocate in paesi diversi ma con raggio di vendita globale.
Per Comete la soluzione è stata quella di inserire una entry di questa entità per ogni attributo necessiti di essere tradotto dalla lingua originale.
È intuibile come la scelta fatta porti all'esplosione della tabella TRANSLATION. Considerando che i dati contenuti nel database possono essere nell'ordine delle decine di migliaia una traduzione completa rappresenterebbe un'occupazione di un numero di elementi pari al prodotto tra le entry presenti e il numero di attributi dell'entità di appartenenza.

2.2.3.3 Analisi database esistenti: Coming Soon

Come per Comete anche per Coming Soon è stato analizzato il json derivato dal modello relazionale.

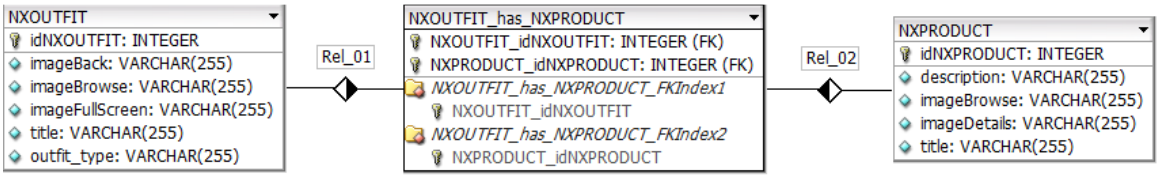

Figura 3: modello relazionale Coming Soon

Lo schema ottenuto è molto ridotto rispetto agli altri, ma contiene una caratteristica inedita: l'utilizzo dell'entità NXOUTFIT. Questa rappresenta un insieme eterogeneo di oggetti utilizzati per realizzare un capo completo.
Inoltre a differenza dello schema relazionale precedente gli allegati non vengono memorizzati in una tabella specifica ma come attributi aggiunti delle entità presenti.
In particolare le entità NXPRODUCT e NXOUTFIT mantengono per ogni entry indicazione delle immagini associate. Dal punto di vista della denormalizzazione questo procedimento consente di mantenere le stesse informazioni in un numero di entità minore. Tuttavia non consente l'inserimento di allegati generici come per esempio video o file musicali e soprattutto ogni immagine deve essere relativa ad una determinata istanza.
L'ultimo vincolo citato non permette di definire degli allegati generici relativi all'intero datastore e all'applicazione come per esempio un'immagine di default relativa ad un'anteprima non disponibile o un video iniziale avviato al lancio del programma sul client. Questa restrizione per gli obiettivi richiesti non dovrà presentarsi nel database finale.

2.2.3.4 Analisi database esistenti: Diesel Kid

Lo studio della base di dati relativa a Diesel Kid è stato più semplice ed immediato in quanto uno schema del modello relazionale era già disponibile e viene riportato di seguente:

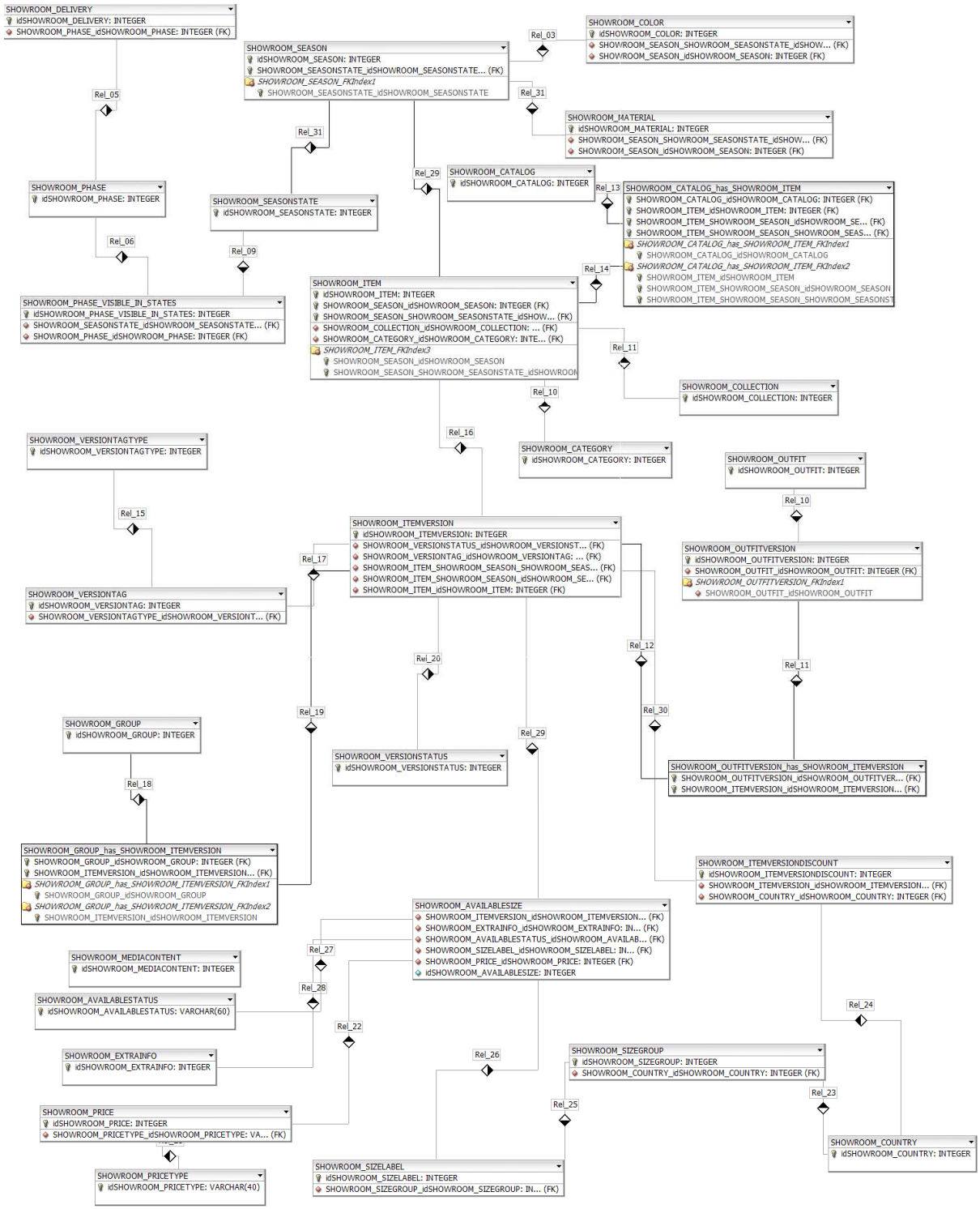

Figura 4: modello relazionale Diesel(gli attributi sono stati omessi per aumentare la leggibilità)

Questo modello è molto interessante in quanto permette di evidenziare diversi aspetti positivi e negativi, in primo luogo sono stati valorizzati i dettagli relativi alle consegne del prodotto e alla taglia. Nonostante le informazioni relative a quest'ultima fossero presenti anche in Comete in questo caso viene assegnato il riferimento anche al gruppo e al paese di appartenenza in quanto ogni taglia, a seconda del paese di fabbricazione del prodotto, assume un significato diverso. Da notare inoltre che un singolo prodotto può avere un'unica taglia descritta in modi distinti.

In questo caso il modello riportato presenta un difetto. È impossibile infatti assegnare ad un oggetto un insieme di etichette distinte relative alla stessa misura in quanto la relazione tra SHOWROOM_AVAILABLE_SIZE e SHOWROOM_SIZE_LABEL è 'molti a uno' e quindi ad ogni taglia disponibile corrisponde al più un'etichetta.
Una possibilità sarebbe quella di assegnare l'etichetta con un protocollo tale da rendere possibile l'immissione di più taglie, per esempio inserendo un'unica stringa contenente *n* etichette separate da un carattere particolare. La soluzione non trova però corrispondenza nel riferimento al gruppo di appartenenza. La relazione che lega SHOWROOM_SIZE_LABEL e SHOWROOM_SIZE_GROUP è infatti ancora 'molti a uno'. Non è quindi possibile esplicitare una serie di gruppi per etichetta.
Altro problema legato alla realizzazione di questo database risiede nella distribuzione degli attributi tra le entità relative al catalogo. Per esempio SHOWROOM_COLOR e SHOWROOM_MATERIAL sono entrambe associate a SHOWROOM_SEASON. La prima contiene informazioni sui colori presenti e sui tessuti associati a determinati colori, la seconda sui tessuti presenti e sui colori associati ai tessuti. Inoltre queste informazioni sono presenti anche in SHOWROOM_ITEMVERSION.
Il database contiene quindi attributi molto ridondanti che ne rendono difficile la gestione. Va inoltre aggiunto che le entità citate sono delle entità deboli in quanto tessuti differenti di uguale colore o colori differenti dello stesso tessuto implicano l'occupazione di righe differenti contrariamente alla denominazione stessa delle tabelle. Non è quindi definibile alcun attributo che le distingua univocamente al di fuori dell'id numerico, caratteristica che le rende di fatto delle entità deboli.
L'aspetto maggiormente interessante dal punto di vista progettuale risiede però nell'esistenza di tag relativi all'entità ITEM_VERSION. Questi rappresentano dei parametri che possono essere utilizzati per qualsiasi attributo aggiunto alla singola entry dell'entità, l'utilità è data dal fatto che gli attributi aggiunti non devono essere omogenei per tutte le entry e quindi questo comporta ad una maggiore efficienza nell'elaborazione delle query da parte del dbms.
Questa caratteristica rappresenterà il filo conduttore del modello finale in quanto incapsula tutti i concetti definiti negli obiettivi iniziali, come verrà spiegato successivamente.

2.2.3.5 Analisi database esistenti: Giesse

L'analisi di questo database è partita dai file csv contenenti i dati del catalogo, e, come in Armani, sono stati utilizzati degli strumenti specifici per analizzare le migliaia di righe contenute in essi.

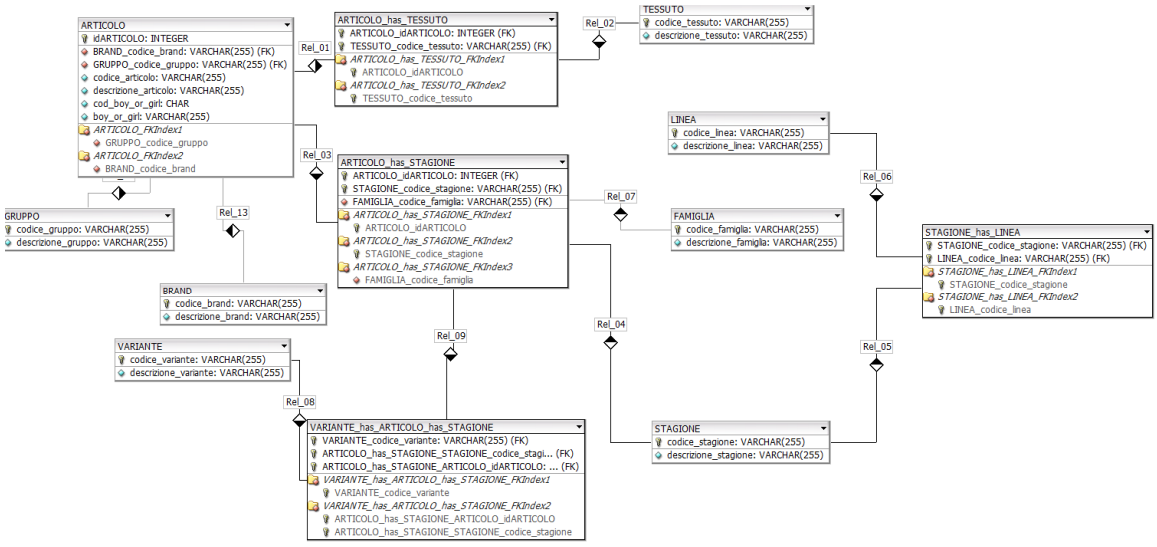

Figura 5: modello relazionale Giesse

Non ci sono delle caratteristiche rilevanti dal punto di vista ingegneristico a meno degli attributi utilizzati ‘gruppo’ e ‘famiglia’. Ancora una volta è utile far notare che gli attributi inediti presenti in realtà hanno lo stesso significato di altri appartenenti ai modelli visti precedentemente.

2.2.3.6 Analisi database esistenti: Panerai

Come per Comete e GS la sorgente di analisi è stato il file json contenente i metadati, da questo è stato possibile ricavare il seguente schema.

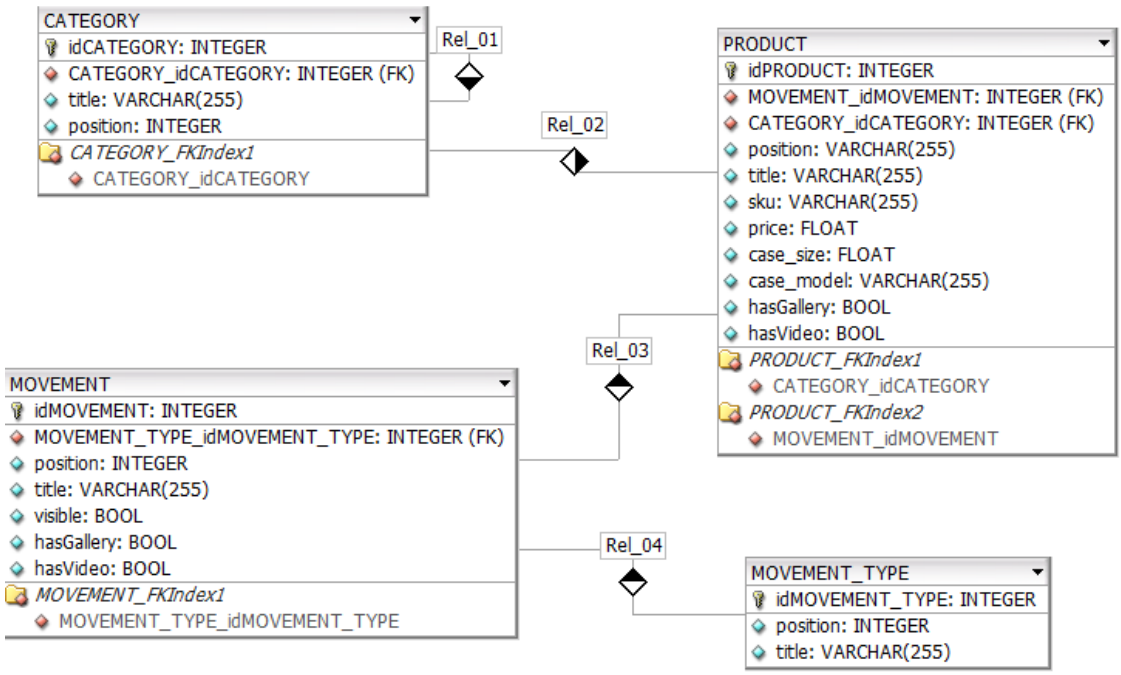

Figura 6: modello relazionale Panerai

Si distinguono alcuni aspetti molto interessanti, dall’utilizzo di una associazione ricorsiva nell’entità CATEGORY all’utilizzo dell’entità MOVEMENT contenente attributi relativi alla rappresentazione grafica del movimento degli oggetti. L’aspetto grafico è infatti fondamentale per il target commerciale di questi prodotti. Inoltre nella libreria di sistema dei dispositivi Apple è disponibile una ricca collezione di oggetti che permettono la visualizzazione dei movimenti e l’interazione con l’utente. Attributi che descrivano una particolarità dell’oggetto in queste rappresentazioni sono da considerarsi quindi una caratteristica aggiuntiva molto importante.

2.2.4 Creazione del modello globale

2.2.4.1 Creazione del modello globale: prima versione

Dopo aver esaminato i database precedenti sono stati evidenziati i seguenti punti sui modelli relazionali:

- Contengono un numero di attributi legati all’aspetto qualitativo degli oggetti molto elevato.
- Alcuni attributi hanno un nome specifico della ditta a cui appartengono ma globalmente rappresentano la stessa proprietà.

- La base di dati deve rappresentare molteplici aspetti legati al catalogo, alla vendita, al cliente, all'agente e agli outfit.
- Il mapping degli attributi da qualsiasi modello esaminato al modello in fase di sviluppo deve essere semplice ed immediato.
- Le elaborazioni prodotte sono relative a dei database parzialmente o totalmente normalizzati.

Come prima versione del modello, che da questo punto in poi verrà chiamato Global, è stato pensato di includere tutti gli aspetti comuni dei database esaminati raccogliendo tutti gli attributi, compresi quelli appartenenti ad un'unica azienda e inserendoli nelle entità specificate ottenendo lo schema relativo(fig.7).
È stato scelto di illustrare una versione già denormalizzata. In realtà le prime ipotesi comprendevano molte più associazioni soprattutto per l'entità Item ma non è stato ritenuto necessario includerle nella discussione.

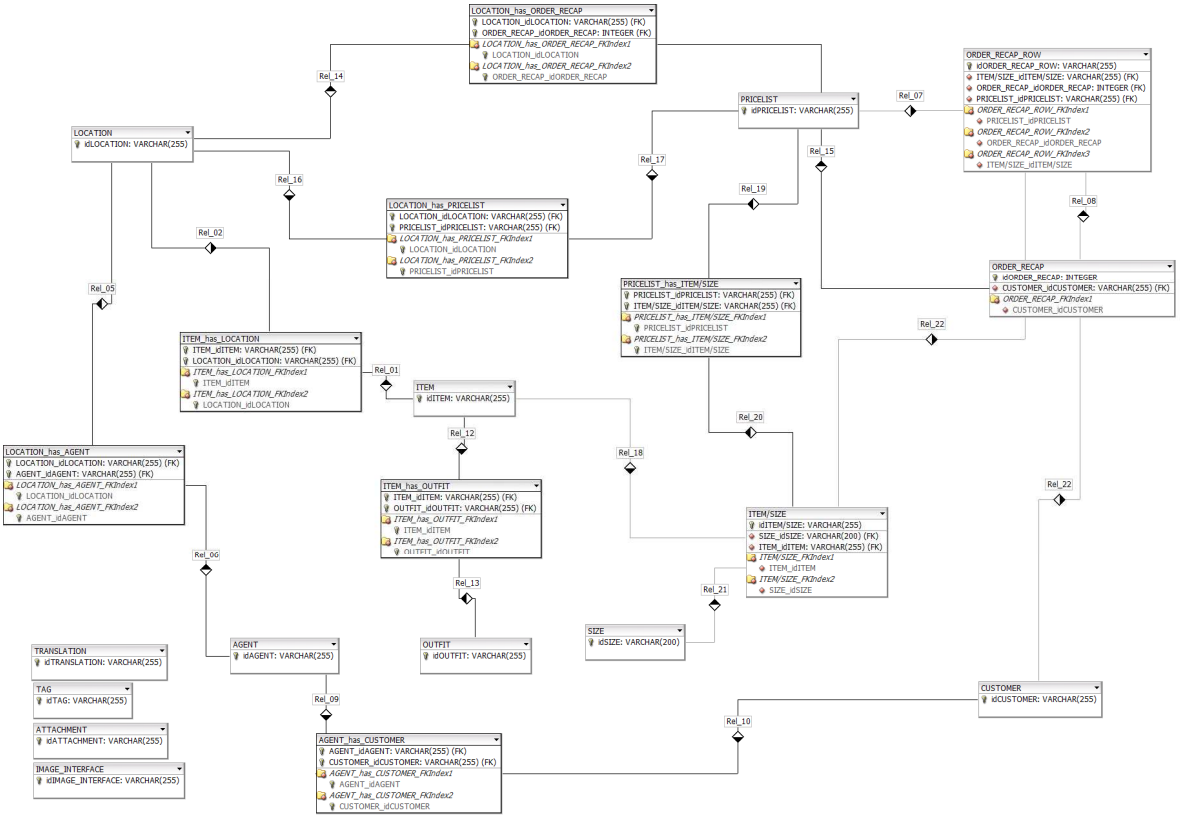

Figura 7 Global prima versione(gli attributi sono stati omessi per aumentare la leggibilità)

La discussione procede ora nella spiegazione dettagliata di ogni entità, dagli attributi di cui è composta e dal significato delle relazioni in cui è coinvolta.

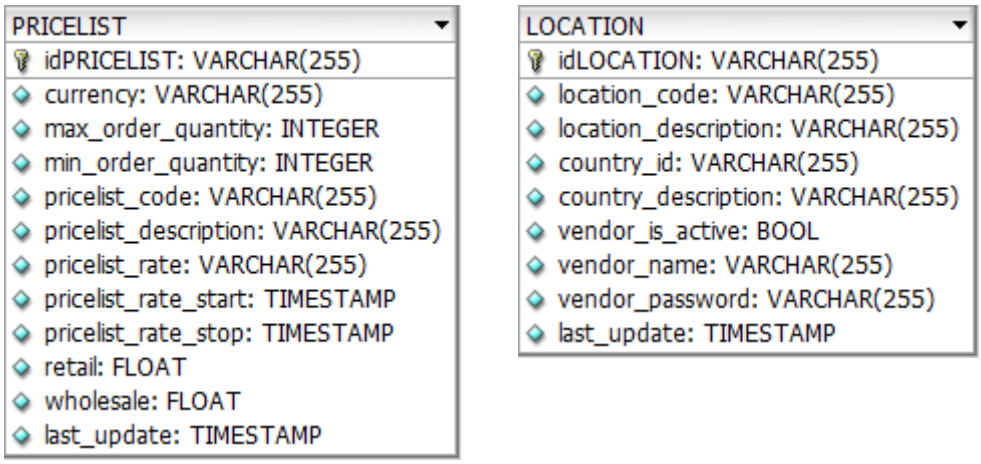

Figura 8 Entità PRICELIST e LOCATION

PRICELIST rappresenta il listino prezzi. Ogni zona commerciale ne detiene uno associato con una valuta riportata e un possibile ordinativo minimo di merce ordinabile oltre agli sconti per la vendita all'ingrosso.
Il listino prezzi è collegato all'ordine effettuato attraverso una relazione 'uno a molti', con i dettagli dell'oggetto e con il venditore è collegato, invece, con una relazione 'molti a molti'. Questo implica che un venditore possa avere più di un listino, ipotesi che verrà discussa nuovamente nel corso delle versioni successive.
LOCATION rappresenta la zona, può essere descritta come una regione geografica, una città, uno stato o anche un concessionario.
Inizialmente in questa prima versione l'entità contiene informazioni personali riguardo ai venditori presenti e indicazione dello stato delle attività. Ogni agente di vendita è collegato ad una o più entry di questa entità, stessa cosa vale per gli oggetti, i listini e gli ordini. Tutte queste relazioni sono 'molti a molti'.
TRANSLATION rappresenta la tabella contenente le traduzioni delle singole entry presenti nel database, per ogni entry della tabella sono presenti i seguenti attributi: *translation* ovvero la traduzione di una determinata parola, *language* la lingua di traduzione e *key* l'identificativo dell'entità di riferimento concatenata al nome dell'attributo e all'id dell'entry.

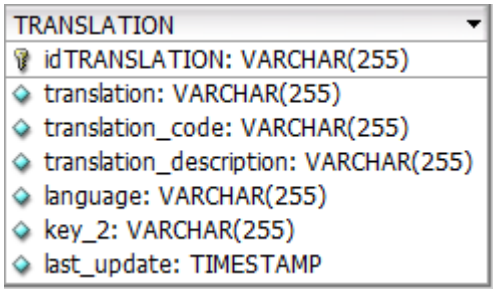

Figura 9 entità TRANSLATION

Questa entità è stata implementata e gestita nell'ambito dello sviluppo della sezione successiva. Nonostante questo non è ancora stato trovato un metodo per la rappresentazione che risponda ai requisiti aziendali e di efficienza forniti.
Una possibile soluzione è quella di creare una base di dati differente per lingua di traduzione in modo tale da rendere la ricerca e la lettura dei dati immediata.

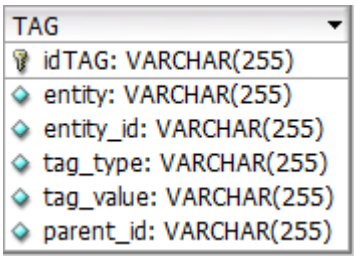

Figura 10 entità TAG

La maggiore innovazione portata dallo schema relazionale sviluppato è data dalla possibilità di inserire *tag*. Come è stato descritto nell'elaborazione effettuata per Diesel Kid essi sono delle stringhe generiche associabili ad una o più determinate entry.
Su questa tecnologia sono stati sviluppati degli studi[3] per verificarne l'adattabilità ad una base di dati. I risultati sono stati soddisfacenti in quanto l'inserimento di queste stringhe generiche permette una categorizzazione automatica e quindi una maggiore velocità nella ricerca. I datastore che utilizzano questa tecnologia, per i motivi elencati, sono sempre più numerosi.
Tuttavia nel corso di questa progettazione è stato riscontrato come i dati forniti debbano necessariamente essere tipizzati. Lo schema costruito, in questo senso, fornisce una soluzione alternativa.
TAG è la tabella di riferimento per la memorizzazione degli attributi dinamici. Per dinamici si intende che il tipo viene specificato real-time senza dover creare nuovamente le tabelle del database e applicato ad un insieme di entry specifiche di una determinata tabella.
In questo modo una singola entry di una tabella può essere associata con un numero arbitrario di parametri non necessariamente coincidente con le altre entry. Per esempio l'elemento della tabella PERSONA *Mario Rossi* può essere associato ai tag *Telefono*, *Cellulare* e *Fax* mentre *Carlo Verdi* può essere associato al solo *Telefono*.
Questo consente di salvare determinate informazioni solo su determinate entry. Attributi che rappresentano la stessa proprietà ma per vincoli aziendali posti dal cliente hanno nomi identificativi distinti possono essere aggiunti semplicemente come tag senza rendere necessaria la duplicazione ridondante degli stessi. Inoltre la dinamicità consente di definire nuove applicazioni in modo semplice ed immediato.
Solitamente i tag sono delle semplici stringhe di valori, questo è efficace se il loro numero è ridotto e quindi è facile definirne un tipo di appartenenza. Nel caso di database con decine di attributi invece è necessario effettuare una tipizzazione dei valori. Sono state analizzate due soluzioni distinte:

- definire ogni tag come oggetto composto dalle variabili di esemplare <value, parent> dove parent è un reference ricorsivo ad un altro tag contenente come valore il nome del tipo dell'istanza puntata.
- definire ogni tag come oggetto composto dalle variabili di esemplare <value, type, parent>. Questa soluzione comporta una maggiore efficienza in fase di ricerca rispetto alla precedente, inoltre risolve il problema degli attributi gerarchici come verrà successivamente descritto.

I campi restanti consentono di specificare l'entità e la singola entry associate nella base di dati.
Il parent, invece, funge da chiave esterna per referenziare un tag con un'altra entry della stessa tabella.
A questo punto una domanda che sorge spontanea riguarda l'utilizzo e la necessità di definire questo attributo.
In realtà per le specifiche di progetto il riempimento del database deve partire da spreadsheet excel o csv ma una futura versione deve prevedere di poter effettuare il riempimento partendo da un altro database e quindi da un insieme di tabelle e riferimenti.

Per questo motivo si è prospettato il seguente problema: se in più tabelle distinte sono stati utilizzati gli stessi identificativi per i campi delle entry, e si desidera che questi nomi restino gli stessi, è necessario mantenere un riferimento con l'entità di partenza in modo tale da evitare ambiguità.
Per esempio supponiamo che nella produzione di un certo abito siano disponibili colori con una percentuale di acrilico variabile: giallo con percentuale 10% e rosso con 90%, e supponiamo inoltre di dover riportare questa informazione nella base di dati.
Memorizzando la percentuale e il colore come semplici tag il risultato sarebbe il seguente:

Id	Entity	entity_id	Type	Value
1	ITEM	1	Acrilico	90%
2	ITEM	1	Colore	Giallo
3	ITEM	1	Acrilico	10%
4	ITEM	1	Colore	Rosso

Come è possibile notare si perde totalmente il riferimento al colore di appartenenza per le differenti percentuali di acrilico.
Per risolvere il problema sono possibili diverse soluzioni: la più semplice è quella di cambiare il tipo del tag in acrilico_giallo e acrilico_rosso in modo da conservare la corrispondenza. Questa soluzione, molto approssimata, non è adatta per i requisiti di adattabilità richiesti.
Per risolvere il problema è stata ideata una soluzione che utilizza il riferimento al tag padre definito dall'associazione ricorsiva nella tabella TAG.

Id	Entity	entity_id	Type	Value	parent_id
1	ITEM	1	Acrilico	90%	4
2	ITEM	1	Colore	Giallo	None
3	ITEM	1	Acrilico	10%	2
4	ITEM	1	Colore	Rosso	None

In questo modo è possibile mantenere le corrispondenze tra i tag. Da notare l'importanza della funzionalità di parent_id nella soluzione scelta.

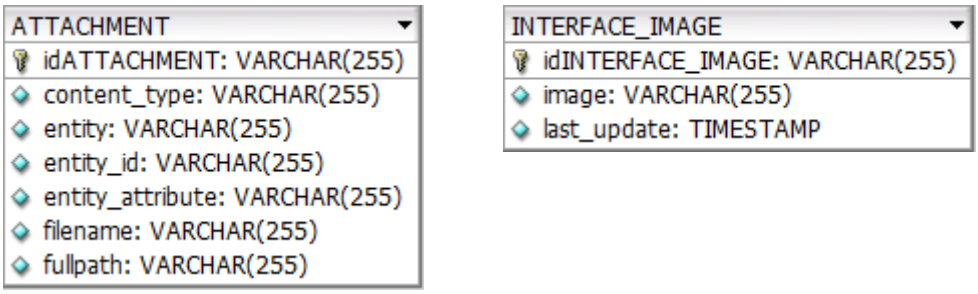

Figura 11 entità ATTACHMENT e INTERFACE_IMAGE

ATTACHMENT è l'entità corrispondente agli allegati, siano essi file musicali, video o immagini. Ogni allegato si riferisce ad una determinata entry di una determinata entità. La gestione dei riferimenti alle entry delle altre entità è la stessa applicata per la tabella TAG.
In realtà in fase di sviluppo la scelta fatta per quest'ultima verrà riconsiderata mentre per gli allegati resterà la stessa per conformità ai requisiti iniziali.
L'allegato è un oggetto particolare in quanto oltre alla memorizzazione degli attributi nella tabella del database deve essere implementato un sistema per l'upload sul server degli stessi attraverso l'indicazione fornita dal campo *fullpath*.
Questo sistema verrà discusso più avanti nel corso dell'implementazione nelle piattaforme centralizzata (Apache) e distribuita(Google App Engine). Per capire l'importanza di questa entità è sufficiente pensare alla quantità di immagini, icone e video presenti in qualsiasi catalogo on-line. La

corretta, semplice e veloce visualizzazione di queste pregiudica sensibilmente la bontà dell'applicazione.
INTERFACE_IMAGE è la tabella contenente le informazioni su immagini relative all'intero catalogo e all'intera base di dati. Non è quindi relativa ad una specifica entry come intuibile dalla mancanza di un attributo relativo all'entità e all'entry associate.
In modo più immediato essa rappresenta una generalizzazione di ATTACHMENT.
L'entità INTERFACE_IMAGE è stata riportata per coerenza con i modelli esaminati. Per questo motivo verrà implementata a livello di base di dati ma mai utilizzata a livello applicativo in quanto superata dall'adattabilità dell'entità relativa agli allegati generici.
TRANSLATION, TAG, ATTACHMENT e INTERFACE_IMAGE non sono collegate attraverso una qualsiasi relazione ad alcuna delle altre tabelle in quanto per le prime tre si tratta di entità che dovrebbero essere associate a tutte le altre. Gli allegati per esempio possono essere relativi all'oggetto come foto di catalogo, ma anche al cliente come la fotocopia della carta d'identità, è quindi necessario siano collegati a tutte le entità. Per questo motivo è stato scelto di realizzare l'associazione a livello più alto piuttosto che assegnare il lavoro al dbms.
Per le prime tre viene specificata l'entità, l'id e nel caso della traduzione anche l'attributo relativo al valore tradotto come illustrato. Nell'ultimo caso non sono necessarie queste informazioni per i motivi specificati.
In realtà a livello implementativo sono state considerate altre soluzioni che permettono di ottenere lo stesso obiettivo specificando gli attributi di riferimento diversamente.

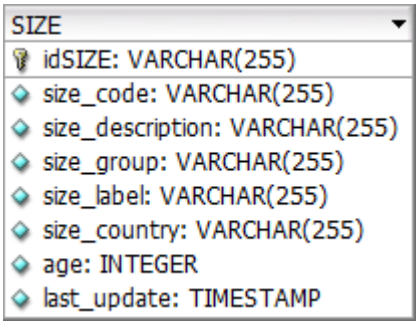

Figura 12 entità SIZE

SIZE specifica gli attributi di ogni possibile unità di misura.
Ogni paese definisce le proprie misurazione ed inoltre sono presenti casi in cui è presente una caratterizzazione per fasce d'età.
È stata realizzata un'entità specifica per la taglia in quanto in un mercato globalizzato come quello attuale è necessario prevedere un'esportazione del prodotto in un numero sempre maggiore di paesi.
Questa entità è però collegata ad ITEM/SIZE da una relazione 'uno a molti'. Il problema descritto nel corso dell'analisi del database Diesel si presenta quindi nuovamente in quanto non è possibile associare una serie di etichette distinte alla stessa taglia.

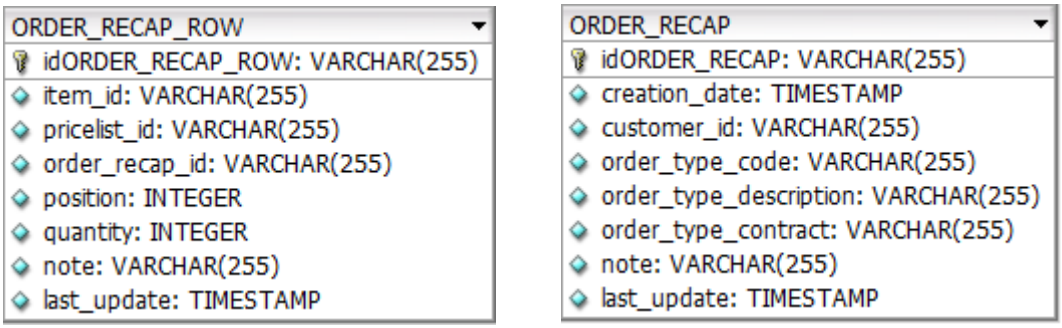

Figura 13 entità ORDER_RECAP e ORDER_RECAP_ROW

In realtà nelle versioni successive verrà descritto come l'utilizzo dei tag permetta di risolvere facilmente questo problema.
ORDER_RECAP definisce invece l'ordine di uno o più prodotti.
Ogni ordine deve essere compilato con le informazioni del cliente che lo effettua, del negozio e dell'oggetto o degli oggetti relativi. Anche nella gestione di questa entità è possibile notare delle inconsistenze, infatti l'ordine avviene tra un cliente e un agente ma non c'è alcuna relazione tra l'entry relativa all'ordine e l'agente che se ne occupa.
Come specificato nella descrizione dell'entità precedente, nell'ordine devono essere presenti i riferimenti ai prodotti associati. Questo in primo luogo era stato risolto attraverso una semplice relazione 'molti a molti' con ITEM/SIZE. La soluzione, sicuramente efficace, non consentiva di formalizzare un ordine con un oggetto distinto per riga. Per questo motivo è stata introdotta l'entità ORDER_RECAP_ROW. Essa rappresenta la riga dell'ordine, e mantiene traccia della quantità relativa.
Un ordine è quindi composto da una o più entry di ORDER_RECAP_ROW. Mentre ciascuna di queste appartiene ad uno specifico ordine.

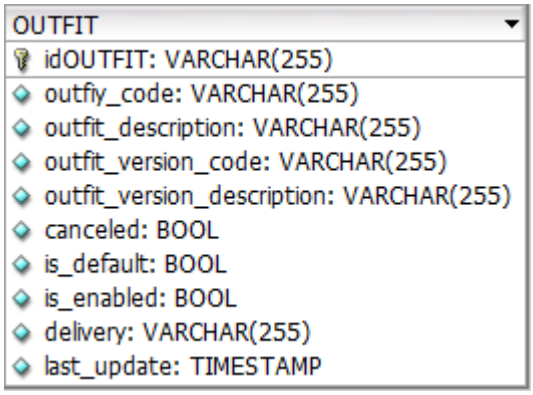

Figura 14 entità OUTFIT

Per ordinare più oggetti dello stesso modello è sufficiente specificare la quantità desiderata nel campo *quantity*. Questa procedura permette di ottenere un insieme di righe distinte e separare quindi gli ordini per oggetto.
OUTFIT rappresenta i gruppi di oggetti che vengono predisposti al fine di presentare le nuove collezioni e di vendere maggiori prodotti possibili. Essendo molto importante dal punto di vista del marketing è stato deciso di inserirla tra le entità principali.
CUSTOMER è l'entità relativa al cliente, fattore distintivo è rappresentato dall'utilizzo di attributi relativi alla residenza e al domicilio fiscale. Ogni cliente è collegato a uno o più ordini e può avere uno o più agenti.
Gli agenti sono descritti nell'entità AGENT, speculare di CUSTOMER con la differenza di non essere legata direttamente agli ordini ma al cliente e al prezzo di listino.
ITEM è l'entità principale, è composta da decine di attributi in quanto è stato scelto un approccio denormalizzato, ideale per l'efficienza ma pessimo per la lettura, per questo motivo è stato deciso di non riportarne l'immagine relativa ma di procedere con una spiegazione testuale.
Essa rappresenta ogni modello di oggetto, ovvero ogni oggetto arricchito da tutte le caratteristiche che lo descrivono qualitativamente quali per esempio il colore, la collezione, l'ordinamento merceologico. Non fanno parte dell'entità descritta invece le caratteristiche che definiscono il singolo elemento quali il codice seriale, la taglia e il prezzo.
Per i dettagli è stata creata l'entità ITEM/SIZE, collegata ad ITEM da una relazione 'molti a uno'.
A primo impatto quest'ultima sembrerebbe l'entità centrale nella mappatura concettuale della base di dati. L'affermazione è vera per quanto riguarda la gestione degli outfit. Per gli ordini e i pagamenti, tuttavia, l'entità di riferimento è ITEM/SIZE in quanto descrive il singolo oggetto fisico e non una classe generica di oggetti. Per questo motivo essa è collegata a ORDER_RECAP_ROW, l'ordine singolo infatti necessita del codice oggetto che è dipendente dalla taglia considerata.

È inoltre collegata a PRICELIST in quanto è possibile avere taglie diverse dello stesso oggetto in listini diversi.
Per ogni taglia di un determinato oggetto è possibile associare uno sconto aggiuntivo rispetto a quello di listino.

2.2.4.2 Creazione del modello globale: Seconda versione

La prima versione comprende in modo completo tutti i concetti definiti. Tuttavia presenta alcune problematiche sensibili, alcune sono già state elencata nella discussione, altre non riguardanti direttamente le relazioni vengono ora descritte.
In primo luogo il numero di attributi dell'entità ITEM dovuto alla denormalizzazione è eccessivo ed inoltre molti sono ridondanti in quanto specifici di alcune case di moda.
Inoltre PRICELIST non è un'entità solida perché in alcuni spreadsheet relativi ai clienti dell'azienda è stata riscontrata la presenza di oggetti che non si trovano in alcun listino.
Gli sconti vengono memorizzati relativamente al listino, alla taglia, al cliente e all'ordine. Questo fattore porta inevitabilmente a della ridondanza in quanto spesso gli sconti che vengono applicati sono gli stessi, quindi in ottica di aggiornamento delle entry ogni sconto da modificare porterebbe alla modifica di un grandissimo numero di record.
In conclusione a queste osservazioni è stata sviluppata la seconda versione.
In primo luogo è stata tolta l'entità PRICELIST e rimpiazzata con PRICE, quest'ultima eredita tutti gli attributi di PRICELIST meno ovviamente quelli indicanti il listino di appartenenza *pricelist_code* e *pricelist_description*.
Nel caso sia presente un listino questo rappresenterà una entry in LOCATION. Date le caratteristiche richieste infatti questa entità è associata al listino più che alla locazione fisica del venditore.
Concettualmente essa deve essere considerata come tale o come il venditore stesso nel caso un oggetto non abbia listini di riferimento. È stata compiuta quindi una fusione concettuale tra PRICELIST e LOCATION separando dalla nuova entità ottenuta gli attributi riguardanti il prezzo di vendita e le possibili opzioni in PRICE.
Inoltre sono state aggiunti per tutte le entità gli attributi *valid_from*, *valid_to*, *note*. L'utilità di questi è data dalla possibilità di aggiornamento intelligente, è possibile infatti evitare di scaricare tutte le entry di ciascuna entità effettuando l'operazione solo per quelle in cui l'istante temporale del download è incluso nell'intervallo [*valid_from*,*valid_to*].

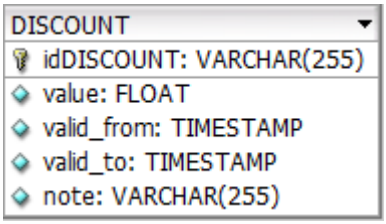

Figura 15 entità DISCOUNT

Essendo il target di client finale un dispositivo mobile come l'IPad un download mirato dei dati dal server rappresenta una caratteristica molto importante.
Rispetto alla versione precedente è stata aggiunta anche l'entità DISCOUNT, che rappresenta tutti gli sconti applicabili al singolo oggetto.
Tale tabella è collegata con le entità relative ai clienti, agli oggetti, alle righe dell'ordine, agli agenti e al prezzo. Mentre le prime quattro definiscono associazioni per gli sconti applicabili la quinta è relativa al prezzo di vendita al quale deve essere applicato lo sconto.
In CUSTOMER sono stati tolti gli attributi relativi al domicilio fiscale del cliente ed è stata creata un'entità di supporto ADDRESS in relazione 'molti a uno' con CUSTOMER. Inoltre è stata aggiunta la relazione tra AGENT e ORDER_RECAP in quanto un cliente può avere più agenti di vendita ma nell'ordine è necessario specificare la locazione fiscale di un unico agente. Essendo la relazione tra

ORDER_RECAP e CUSTOMER ‘molti a uno’ e tra CUSTOMER e AGENT ‘molti a molti’ l’informazione relativa all’ordine effettuato non è quindi rintracciabile. È necessaria per questo motivo una relazione diretta tra AGENT e ORDER_RECAP.
Anche la gestione degli ordini è stata modificata. Nella versione precedente ogni riga ordine, rappresentata da una entry in ORDER_RECAP_ROW, era collegata a PRICELIST e ITEM/SIZE in modo tale da mantenere il collegamento con l’oggetto e il listino contenente il prezzo.
In questa versione il riferimento all’oggetto passa attraverso una terza entità: ORDER_TEMPLATE_ITEM, che collega le righe ordine con i dettagli oggetto specificandone la quantità ordinata. Questa entità in particolare contiene dei template preconfigurati per l’acquisto degli oggetti.

2.2.4.3 Creazione del modello globale: Terza versione

La seconda versione ha introdotto diversi miglioramenti ma il progetto ancora non soddisfa i requisiti in quanto in primo luogo DISCOUNT è collegata ad ITEM e non ad ITEM/SIZE ma per un dato oggetto le taglie differenti possono subire sconti differenti.
Inoltre in ottica di efficienza e di servizi client-side data una specifica soglia di prezzo per risalire a tutti gli oggetti al di sotto di tale valore sono necessari tre join. Essendo l’obiettivo quello di garantire le operazioni di ricerca efficienti è opportuno che almeno le operazioni più frequenti(come la ricerca per prezzo) soddisfino questo requisito.
Per questo motivo è stata sviluppata la terza versione, che differisce dalla seconda solo per la denormalizzazione compiuta nelle tabelle relative ai dettagli dell’oggetto:

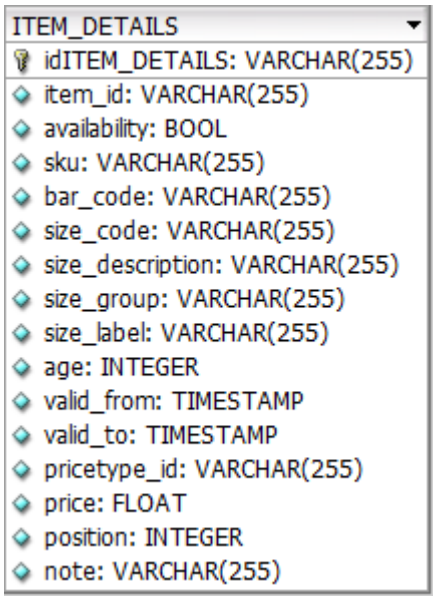

Figura 16 entità ITEM_DETAILS

Il miglioramento introdotto da questa versione risiede infatti nell’utilizzo di una nuova entità che accorpa ITEM/SIZE, PRICE e SIZE: ITEM_DETAILS. Non era efficiente mantenere separate queste entità in quanto contengono attributi direttamente collegabili agli item e quindi molto utilizzati in fase di ricerca. Tutti i dettagli relativi al prezzo differenti dal prezzo stesso sono memorizzati invece in PRICE_TYPE, tabella in relazione ‘uno a molti’ con ITEM_DETAILS.
La tabella illustrata è collegata con una relazione ‘molti a uno’ con ITEM e consente quindi di effettuare query di ricerca per prezzo di listino in modo molto più efficiente rispetto alla soluzione presentata nella versione precedente.

Inoltre il problema relativo alle informazioni associate ad etichette diverse della stessa taglia viene risolto utilizzando la tabella TAG. Ad ogni entry di ITEM_DETAILS vengono associate infatti *n* entry di TAG dove *n* rappresenta il numero di etichette disponibili. Ciascuna entry sarà composta dal nome dell'etichetta e da un tipo che ne specifica il paese di conformità.
A questo punto la struttura fondamentale è stata creata, resta da gestire la questione delle entità composte da decine di attributi come ad esempio CUSTOMER o ITEM.
Infatti, come già specificato precedentemente, il modello relazionale è stato ottenuto dallo studio di tutti gli altri. Essendo i modelli analizzati molto diversi, soprattutto per i nomi associati agli attributi, le tabelle inserite presentano dei riferimenti molto ridondanti.
La soluzione a questo problema è data dalla tabella TAG, il cui utilizzo non è stato ancora illustrato in modo dettagliato. È stato deciso di partire infatti dalla spiegazione del modello fisico dei dati e quindi dal punto di vista del DBMS. La gestione degli attributi dinamici invece viene realizzata ad alto livello e, fino a questa versione del modello, non erano molti gli attributi che era stato deciso di non includere nelle tabelle ma di inserire come entry della tabella TAG, come per esempio le etichette relative alla taglia degli oggetti.

2.2.4.4 Creazione del modello globale: quarta versione

La versione successiva in questo presenta un forte cambiamento in quanto è stato deciso di inserire tutti gli attributi, meno quelli basilari, come tag.
In questo modo è stato possibile ottenere uno schema del database molto più omogeneo dal punto di vista delle realtà attuali dell'azienda. Questo era infatti il requisito principale del lavoro, ovvero un modello adatto ad ogni realtà e da utilizzare come base per sviluppi successivi.

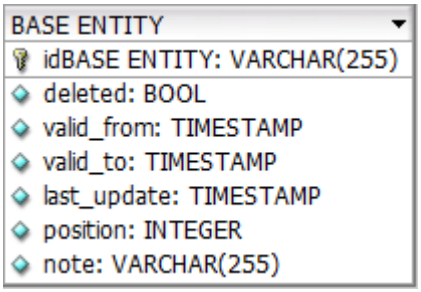

Figura 17 entità BASE ENTITY

In primo luogo è stata creata un'entità base dalla quale tutte le altre ereditano gli attributi: BASE ENTITY(fig. 17).
L'entità citata racchiude tutte le caratteristiche che consentono di determinare lo stato dell'entry associata. Partendo dal primo attributo, *deleted* indica se l'entry relativa non è più utilizzabile fungendo quindi da indicatore per un possibile *pruning* del database, ovvero per la cancellazione delle informazioni obsolete seguita dal controllo della consistenza dei riferimenti associati.
Gli attributi *valid_from*, *valid_to* e *last_update* sono indicatori dello stato di obsolescenza dell'entry mentre *position* indica la priorità dell'entry rispetto alle altre nella tabella e *note* rappresenta uno spazio per le notazioni generiche di sincronizzazione.
Le relazioni sono rimaste le stesse della versione precedente, sono cambiati totalmente, invece, gli attributi di ogni singola entità. ITEM da quaranta attributi è passata a quattro: *code*, *description*, *model_code*, *model_description*. Attributi che definiscono rispettivamente il codice relativo alla versione specifica e il modello dal quale è stata ricavata la versione stessa.
CUSTOMER da venti attributi è passata a sei: *code*, *description*, *name*, *surname*, *language*, *vat_code*.
Stessa cosa è avvenuta per AGENT, inoltre sia LOCATION che OUTFIT sono ora composte dai soli *code* e *description*. È avvenuta quindi una drastica riduzione nel numero di attributi per entità, inoltre quelli rimasti ne rappresentano la descrizione essenziale e quindi sono adatti a qualsiasi modello esaminato precedentemente.

Gli attributi che sono stati tolti dal modello relazionale vengono mappati ad alto livello come entry della tabella TAG. Per esempio *collection_code* e *collection_description* sono stati tolti da ITEM ma per ogni oggetto saranno presenti due tag con *type* 'collection_code' e 'collection_description' e *value* il valore attribuito inizialmente.
In questo modo è facile intuire come la tabella TAG conterrà un numero sproporzionato di entry rispetto alle altre. Come verrà spiegato nel corso dell'implementazione, il tempo necessario a salvare tutti questi dati rappresenterà un parametro sensibile in quanto gli accessi multipli in memoria di massa consistono in operazioni molto lente dal punto di vista computazionale.
TAG, ATTACHMENT, INTERFACE_IMAGE e TRANSLATION invece non hanno subito alcuna riduzione in quanto non sono specifiche di nessuna casa di moda particolare ma vengono utilizzate come supporto per i dati inseriti.
È cambiata radicalmente anche la gestione degli ordini, infatti non viene più utilizzata l'entità ORDER_TEMPLATE per il collegamento tra l'oggetto e l'ordine relativo ma tale riferimento viene realizzato attraverso la relazione 'uno a molti' tra ITEM_DETAILS e ORDER_ROW. Infatti ORDER_TEMPLATE non era un'entità solida in quanto non è detto che tutti gli oggetti dispongano di ordini precompilati, la gestione di questi ultimi passa ora attraverso ORDER e TAG. Gli ordini precompilati vengono memorizzati nella tabella ORDER utilizzando un attributo *isTemplate* come indicatore della caratteristica. Questo attributo come molti altri viene modellato come tag.
DISCOUNT inoltre è ora collegato con una relazione 'molti a molti' sia a ORDER che a ORDER_ROW.

2.3 Conclusioni

Partendo dall'analisi dai dati forniti, relativi a clienti distinti è indipendenti, è stata creata una struttura completa ma soprattutto adattabile.
La base di dati sviluppata è in grado di contenere tutti gli attributi presentati nel corso dell'analisi iniziale limitando tuttavia la memorizzazione fissa a quelli essenziali.

Questo passo, fondamentalmente teorico, rappresenta la base dell'applicazione generatrice oggetto dei capitoli successivi.

Capitolo 3

Web app generatrice su architettura Django/Apache

3.1 Background

3.1.1 Orm

La maggior parte delle applicazioni definite secondo il paradigma della programmazione ad oggetti utilizza un database relazionale per le necessità di memorizzare i dati in modo persistente.
Tuttavia, i paradigmi relazionale e orientato agli oggetti rappresentano la stessa informazione in locazioni differenti, il primo in memoria di massa, il secondo in memoria centrale.
La separazione illustrata implica, nell'implementazione dell'applicazione, la creazione di due differenti modelli per la memorizzazione e la gestione degli stessi dati, aumentando quindi la difficoltà e il tempo necessario.
ORM(Object Relational mapping)[4] è una tecnica di programmazione per la gestione semplificata dei sistemi RDBMS. In particolare un framework che supporta questa tecnica fornisce un livello di collegamento tra il modello ad oggetti definito nell'applicazione, detto *modello logico*, e il modello relazionale definito nel RDBMS, detto *modello fisico*.
Utilizzando questa tecnologia è sufficiente definire il modello logico per derivare automaticamente il modello fisico equivalente. Inoltre la gestione stessa delle operazioni di inserimento, modifica, cancellazione e ricerca può essere implementata a livello applicativo in quanto la traduzione in operazioni di basso livello verrà effettuata dal livello intermedio ORM.
Non è necessario quindi implementare manualmente la struttura del database. Inoltre vengono forniti una serie di servizi di supporto per l'ottimizzazione delle operazioni compiute utilizzando caching in memoria delle query più frequenti e strutture complesse per la derivazione dei dati rendendo quindi più semplice e intuitivo lo sviluppo delle applicazioni rispetto alle specifiche in linguaggio SQL.

Per esempio la creazione di un oggetto dell'entità ITEM avviene nel seguente modo(nell'esempio viene utilizzato il supporto ORM fornito dal framework django, che verrà presentato nella sezione successiva):

```
i=Item(code='0001', description='jeans Armani')
i.save()
```

mentre l'equivalente in linguaggio SQL:

```
INSERT INTO ITEM
VALUES (id=1, code='0001',description='jeans Armani')
```

Il fetching di tutti gli elementi della tabella ITEM invece viene ottenuto attraverso la seguente istruzione:

```
Item.all()
```

al posto di:

```
SELECT * FROM ITEM
```

Supponendo di voler recuperare tutti gli item_details relativi ad un certo item è sufficiente richiamare il metodo seguente:

```
i.item_details_set.all()
```

al posto di:

```
SELECT * FROM ITEM_DETAILS JOIN ITEM
WHERE ITEM_DETAILS.id =ITEM.outfit_id
```

Un livello intermedio per mappare le istruzioni ad alto livello in un linguaggio compatibile con il dbms porta all'indipendenza nello sviluppo dell'applicazione dato il gestore della base di dati utilizzato e a una maggiore leggibilità del codice.
Nella filosofia della programmazione ad oggetti inoltre uno sviluppo dell'applicazione con un unico livello di logica relativo al mapping degli oggetti rappresenta un risultato positivo per la stratificazione e la modularità.

3.1.2 Django

Django[5] è un framework per web application open-source basato sull'architettura model-view-controller scritto interamente in python.

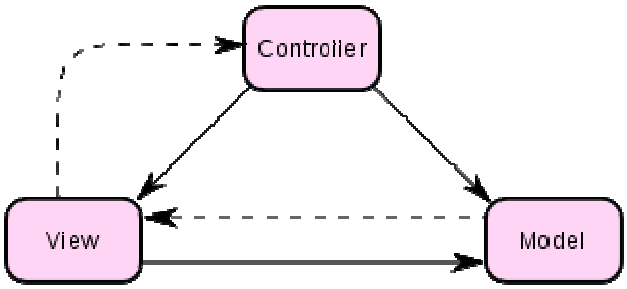

Figura 18 Schema Model-View-Controller

In questa architettura vengono distinte la parte di logica relativa al processing dei dati e quella basata sull'interfaccia utente separando quindi i contesti per quanto riguarda lo sviluppo, il testing e la manutenzione dell'applicazione.
In particolare i moduli relativi sono i seguenti:

- model: gestisce la parte relativa al db, per quanto riguarda quindi l'inserimento, la ricerca e in generale la gestione dei dati.
- view: gestisce la parte relativa alla visualizzazione degli elementi gestiti da model e fornisce supporto per l'interazione da parte degli utenti.
- controller: gestisce le richieste inviate dal client generando le risposte attraverso la creazione di azioni relative agli altri moduli basate sulla richiesta ricevuta.

L'obiettivo principale consiste nella creazione di pagine web complesse e basate su componenti caricati dal database nel contesto della riusabilità e della modularità.
Il framework è composto da un ORM che funge da traduttore dai modelli definiti in django e il database relazionale sul quale i dati vengono effettivamente inseriti, un dispatcher URL basato sulle espressioni regolari in grado di effettuare il parsing degli indirizzi che vengono inseriti secondo delle specifiche anche complesse, un sistema di view per l'elaborazione delle richieste del client ed infine un sistema di template per la visualizzazione dei contenuti.
Django può essere lanciato in esecuzione con Apache utilizzando mod_python o mod_wsgi come protocolli di comunicazione e di trasmissione dati tra client e server. Per quanto riguarda il datastore Django supporta quattro DBMS distinti: PostgreSQL, MySql, SQLite e Oracle.
Esiste inoltre una versione di Django che supporta DBMS non relazionali come per esempio BigTable utilizzato da Google App Engine che verrà presentata nel corso del capitolo quarto.
La versione di Django utilizzata è la più recente, ovvero la 1.2.3, questa comprende delle caratteristiche aggiuntive rispetto alle precedenti ed è perfettamente compatibile con la piattaforma Google App Engine.

3.1.3 Apache e SQLite 3

Apache http Server project[8] è un software sviluppato da un team di collaboratori con l'obiettivo di creare un robusto, solido, efficiente e open-source server web HTTP.
La qualità e l'affidabilità del prodotto non sono inferiori a quelle dei migliori server web attualmente sul mercato.
In particolare Apache supporta una grande varietà di caratteristiche, ciascuna implementata come un modulo compilato che estende le funzionalità di base, fornite dal modulo principale, il *core*.
Le altre funzionalità offerte vanno dal supporto per la programmazione back-end fino agli schemi per l'autenticazione dell'utente.
Apache è il server web maggiormente utilizzato dal 1996. Nel 2005 una ricerca sviluppata da Netcraft Web Server giunse al risultato che il 70 % dei portali utilizzavano Apache.
SQLite[9] è un DBMS relazionale contenuto in un'unica libreria C. Open-source e di facile configurazione, si distingue dai DBMS tradizionali in quanto per le query SQL non viene instanziato un processo separato che viene utilizzato dall'applicazione del server per gestire le richieste del client. Viene invece creato un processo integrato nell'applicazione stessa.
Al contrario dei DBMS tradizionali per la gestione delle basi di dati di applicazioni client/server(e quindi con accesso ai dati remoto) il motore SQLite non ha quindi processi standalone con i quali l'applicazione back-end può comunicare ma semplicemente una libreria collegata all'applicazione front-end.
La mancanza di tale processo riduce la latenza delle chiamate alle funzioni di inserimento nel datastore e nelle comunicazioni interprocesso.
Per questo motivo è molto più veloce rispetto ad altri DBMS. Caratteristica che lo porta ad essere una scelta frequente per quanto riguarda i sistemi embedded, oltre ovviamente per le applicazioni Web.
SQLite inoltre mantiene l'intero database(quindi tabelle, indici e dati) come un singolo file nel server ed è quindi di immediata configurazione. Queste proprietà hanno caratterizzato la scelta compiuta in fase progettuale a favore di questo DBMS.

3.1.4 Python

Python[10] è un linguaggio di programmazione ad alto livello, orientato agli oggetti e utilizzato soprattutto per lo sviluppo di script e applicazioni server-side. A differenza di altri linguaggi di scripting, come Perl o PHP, consente di utilizzare una vasta libreria di funzioni che permettono quindi di sviluppare applicazioni molto complesse. Per la sua semplicità, dinamicità e flessibilità rappresenta il linguaggio di programmazione più utilizzato per web application.
Python supporta diversi paradigmi di programmazione, come quello object-oriented, quello imperativo e quello funzionale, ed offre una tipizzazione dinamica forte.
Le sue caratteristiche principali per quanto riguarda la stesura del codice sono la non tipizzazione delle variabili e lo scope delegato all'indentazione. Questo lo rende un linguaggio semplice da usare e imparare per l'intuibilità dei suoi costrutti logici, chiari e non ambigui.
Esso inoltre è un linguaggio pseudo compilato, un interprete si occupa di analizzare il codice sorgente(descritto in file di testo con estensione .py) e, se sintatticamente corretto, di eseguirlo. Non esiste quindi una fase di compilazione separata che generi un file eseguibile partendo dal sorgente. Questa caratteristica lo rende portabile. Una volta scritto un sorgente, esso può essere interpretato ed eseguito sulla gran parte delle piattaforme attualmente utilizzate. Per una corretta esecuzione è sufficiente, infatti, la presenza della versione corretta dell'interprete.
Non è stata utilizzata l'ultima versione rilasciata, ovvero la 3.0, in quanto è ancora in fase sperimentale. Inoltre essa è incompatibile con Google App Engine, oggetto del prossimo step, che utilizza la versione 2.5.
È stata quindi scelta la versione 2.6.1, la più recente perfettamente compatibile con quella supportata da Google App Engine.

Django è scritto totalmente in python ed inoltre costituisce un framework di alto livello per lo sviluppo di web application definite nello stesso linguaggio. Per questo motivo e per la totale compatibilità con Google App Engine python è stato scelto come linguaggio di programmazione per la parte back-end.

3.1.5 Definizione strutturale della Web application

Una volta installate tutte le componenti elencate nelle sezioni precedenti è possibile proseguire nella definizione della web application da realizzare.
Per prima cosa è necessario creare un progetto django[7]. Questo non è altro che un contenitore di applicazioni. Per questa operazione il framework mette a disposizione uno script apposito richiamabile attraverso l'istruzione *django-admin.py startproject progetto*. L'esecuzione da riga di comando consegue alla creazione della cartella *progetto*, e di file atti alla configurazione posti all'interno:

progetto/
- __init__.py
- manage.py
- settings.py
- urls.py

Il primo file è relativo all'inizializzazione. In questo caso data dal fatto che viene detto all'interprete python che la cartella *progetto* rappresenta un package.
Il file *manage.py* invece è un contenitore di metodi eseguibili da linea di comando utili per esempio per creare una nuova applicazione, per avviare il server o per eliminare ogni entry dal database.
Il terzo file, *settings.py*, rappresenta la configurazione iniziale. In esso sono contenuti i riferimenti per il DBMS da utilizzare con l'eventuale user-name e password, le cartelle dove memorizzare i file temporanei e le applicazioni facenti parte del progetto.
L'ultimo file contiene gli url relativi. Essendo la web app interpretata dal browser web infatti deve essere definito un mapping degli indirizzi necessari. In questo modo il server, alla richiesta di un indirizzo generico, controllerà il match nel file *urls.py* e, se riscontrato, avvierà la *view* specificata. Per *view* si intende un metodo che gestisce le richieste del server fornendo come risposta un output specifico. Ogni view, o vista, riceve quindi in ingresso una richiesta instradata attraverso la lettura del file urls.py restituendo una risposta.
La sintassi definita nel file contenente gli url è la seguente :

(r'espressione regolare', 'vista da richiamare')

La lettera *r* indica che la stringa successiva è un'espressione regolare(la sintassi di espressione regolare viene definita nella librerie di python). Il controllo viene quindi compiuto tra l'indirizzo specificato dall'utente nel browser e ogni set di url definito da un'espressione regolare.
Il primo match riscontrato porta all'esecuzione della vista relativa.
Una volta svolta l'inizializzazione e la configurazione deve essere creata la web application, per fare questo viene utilizzato manage.py attraverso il comando *python manage.py startapp applicazione.*
L'esecuzione crea una nuova cartella *applicazione* interna a *progetto* contenente:

applicazione/
- __init__.py
- models.py
- tests.py
- views.py

Il primo file elencato, *__init__.py*, contiene il metodo che viene richiamato all'esecuzione dell'applicazione dal server. Il secondo, *models.py*, invece rappresenta la descrizione del modello logico.

Il file *tests.py* contiene il codice per eseguire il testing dell'applicazione, il framework fornisce infatti delle librerie per il testing automatico.
Nell'ultimo, *views.py*, infine vengono dichiarate e implementate le viste disponibili.
Il match conseguente alla richiesta inviata dal client e dal recupero della vista dal file urls.py genera l'esecuzione del metodo corrispondente contenuto in *views.py*.
A questo punto è stato creato sia il progetto che l'applicazione. Una domanda che però sorge spontanea, soprattutto agli sviluppatori C++, è la mancanza di un main o comunque di un metodo principale.
In questo caso non è presente in quanto non è necessario. Il framework django fornisce infatti una struttura completa e generica rendendo semplice lo sviluppo di una nuova applicazione senza la creazione di ulteriori classi ma semplicemente seguendo la struttura fornita. In questo caso ogni vista corrisponde ad un main specifico ad una richiesta.

3.1.6 Comunicazione client-server

Django, per la comunicazione tra il client e il server e, in generale, per passare informazioni relative allo stato del sistema, utilizza oggetti relativi alle *request* ricevute e alle *response* inviate[6].
Ogni request corrisponde alla richiesta di una singola pagina con allegate delle possibili informazioni aggiuntive. Come è stato descritto nella sezione relativa alla struttura della web application la richiesta di una pagina o in generale di una vista viene effettuata attraverso la specifica dell'url.
Non è corretto definire una corrispondenza stretta tra l'url e l'oggetto request associato in quanto richieste distinte possono essere associate al medesimo indirizzo(è sufficiente pensare ad una pagina di invio dei dati o dei file).
In ottica di framework django ad ogni richiesta viene creato automaticamente un oggetto *HttpRequest* contenente i metadati necessari per la gestione dell'input. Al caricamento della view l'oggetto creato viene passato come primo argomento e al termine della propria esecuzione la view stessa deve restituire un oggetto *HttpResponse* come risposta.
Viene di seguito riportato un esempio di vista django:

```
def vista(request):
   if request.method=='POST':
         return HttpResponse ("sono stati inviati dei dati attraverso il protocollo POST")
   return HttpResponse("nessun dato inviato attraverso il protocollo POST")
```

L'esempio, nella sua semplicità, rappresenta la struttura essenziale di una generica view. Il parametro *request* è l'oggetto di tipo HttpRequest creato al momento della richiesta della pagina. Questa può essere effettuata dal client ma anche dal server. Infatti per passare da una vista all'altra fornendo i dati relativi allo stato attuale viene comunque generata una richiesta che viene successivamente elaborata dal server stesso.
Dal parametro request è possibile astrarre le informazioni inviate. Per esempio *request.POST* è una variabile che implementa una struttura dati *dizionario* contenente tutte le variabili per cui è stato indicato il trasferimento da una vista all'altra utilizzando il metodo POST passate quando è stata generata la richiesta. Invece *request.GET* contiene tutte le variabili per cui è stato specificato il metodo di trasferimento GET.
Per recuperare i file inclusi nella pagina al momento della richiesta viene inizializzato il dizionario *request.FILE*, mentre per tutti i metadati relativi all'invio della richiesta, come per esempio l'indirizzo IP del client o la porta del server in cui avviene la comunicazione viene utilizzato *request.META*.

Come è possibile notare il framework fornisce un insieme di API che rendono la comunicazione tra il client e il server semplice ed immediata. Anche per questo motivo è maturata la scelta di utilizzo per lo sviluppo di questo progetto.

Al contrario delle richieste ricevute, e quindi degli oggetti HttpRequest, le risposte non vengono create automaticamente dal server. La loro inizializzazione deve essere specificata, infatti, nel codice dallo sviluppatore back-end attraverso la creazione degli oggetti *HttpResponse*.
Il tipico utilizzo è dato dal passaggio del contenuto da visualizzare nella pagina come stringa. Per fare questo viene creato l'oggetto richiamando il costruttore standard inserendo la stringa come primo parametro. Nell'esempio riportato precedentemente la risposta veniva creata esattamente in questo modo.
Utilizzando questo metodo non è però possibile passare informazioni da una vista all'altra. Si supponga infatti di dover richiamare una vista passando dei dati relativi all'elaborazione compiuta per avviarne un'altra.
In primo luogo è possibile effettuare una fusione tra le due view in modo tale non sia necessario il trasferimento dei dati. Essendo però uno degli obiettivi fondamentali la modularità del codice è necessario e fondamentale mantenere separate le viste.
Tra viste distinte non è possibile la condivisione di puntatori in memoria centrale e quindi dei dati allocati. È possibile però richiamare una vista all'interno di un'altra passando gli attributi e i contenuti delle variabili.
A questo scopo sono state implementate delle sottoclassi di HttpResponse che estendono il concetto fondamentale di risposta permettendo di inserire dei dati allegati e di reindirizzare la richiesta verso un'altra view.
Attraverso gli oggetti *HttpResponseRedirect* è possibile infatti specificare l'url della vista desiderata generando la relativa richiesta.
Successivamente viene riportato un esempio di utilizzo:

```
views.py
def prima_vista(request):
   return HttpResponseRedirect('/mysite/seconda_vista/primavista!/')
def seconda_vista(request, stringa):
   return HttpResponse(stringa)

urls.py
(r'^mysite/prima_vista/$', 'mysite.data_app.views.prima_vista'),
(r'^mysite/seconda_vista/(?P<stringa>\w+)/$', 'mysite.data_app.views.seconda_vista')

Input: http://127.0.0.1:8000/mysite/prima_vista/
Output: primavista!
```

A questo punto è stato analizzato il modo in cui viene generata la risposta come contenuto testuale e come redirect verso un'altra vista. Django permette però di inviare e visualizzare i dati dal server verso il client attraverso il browser web utilizzato e il protocollo http utilizzando dei *template*.
È possibile in altre parole realizzare delle pagine web con delle variabili inizializzate al momento dell'invio della risposta da parte del server. La funzione che permette di fare questo è *render_to_response* la cui firma è riportata di seguito:

```
render_to_response(template[,dictionary][, context_instance][, mimetype]).
```

Dove *template* indica il file html da utilizzare come interfaccia per la visualizzazione del contenuto, *dictionary* il dizionario contenente le variabili da passare come argomento, *context_instance* il contesto della risposta e *mimetype* il tipo della stessa.
In realtà può essere ottenuto lo stesso risultato utilizzando l'oggetto HttpResponse e passando al posto di una semplice stringa un tipo di dato astratto costituito dall'insieme di metadati relativi al template e alle variabili da mappare. Il metodo illustrato, infatti, non definisce alcuna funzione particolare ma utilizza un procedimento di visualizzazione del contesto, inserito come argomento nel template specificato, attraverso l'utilizzo di oggetti HttpResponse come illustrato in precedenza.

La funzione presentata rappresenta quindi un metodo immediato che permette di ottenere lo stesso risultato scrivendo molto meno codice.
Per quanto riguarda le potenzialità è possibile renderizzare qualsiasi tipo di variabile utilizzando una serie di tag html forniti da django. Ogni variabile presente nel codice html nella forma *{{variabile}}* viene inizializzata al momento della chiamata di render_to_response attraverso la ricerca della variabile mappata nel dizionario passato per argomento.
Viene di seguito fornito un esempio di chiamata render_to_response:

```
views.py
def prima_vista(request):
  string=”primavista!”
  render_to_response('index.html',{‘stringa’:string},RequestContext(request, {}), None)

index.html
<html>
        <head>
        </head>
        <body>
                {{stringa}}
        </body>
</html>
```

Al momento della generazione della risposta verrà visualizzata la pagina index.html e stampata a video la stringa.
Django mette inoltre a disposizione, come verrà illustrato successivamente, una serie di tag per la gestione e la manipolazione delle variabili.

3.2 Implementazione

3.2.1 Strumenti utilizzati

Lo sviluppo di questa parte è stato compiuta su una macchina Apple Macintosh con sistema operativo iOs 10.6.4, processore Intel Core Duo da 2.4 Ghz e 2GB di memoria DDR2 SDRAM da 667Mhz.
Come IDE per la programmazione è stato utilizzato Eclipse Helios con l’integrazione del plugin per l’interprete python.
Per l’inizializzazione è stato necessario installare Apache 2.2.14 e Django 1.2.3 mentre per SQLite non è stata necessaria alcuna installazione in quanto non necessita di un processo server-side ma di una libreria integrata con python versione 2.5 e successive.
Il linguaggio di programmazione utilizzato è python versione 2.6.1.

3.2.2 Requisiti e obiettivi

L’obiettivo di questo step è quello di ottenere una web application *wizard*. Ovvero un’applicazione in grado di generare altre applicazioni dall’input ricevuto.

Il progetto(il cui schema è riportato in fig. 19) prevede l’inserimento del foglio elettronico contenente tutti i dati relativi ad un determinato cliente. Caricamento che può avvenire sia da locale che da remoto. Successivamente utilizzando il vasto framework offerto da django e l’efficienza del server e del dbms utilizzati i dati contenuti nello spreadsheet inserito vengono mappati nella base di dati globale illustrata nel capitolo due. Il riempimento avviene secondo le clausole stabilite da un utente configuratore.

Successivamente alla configurazione viene generata la nuova applicazione.

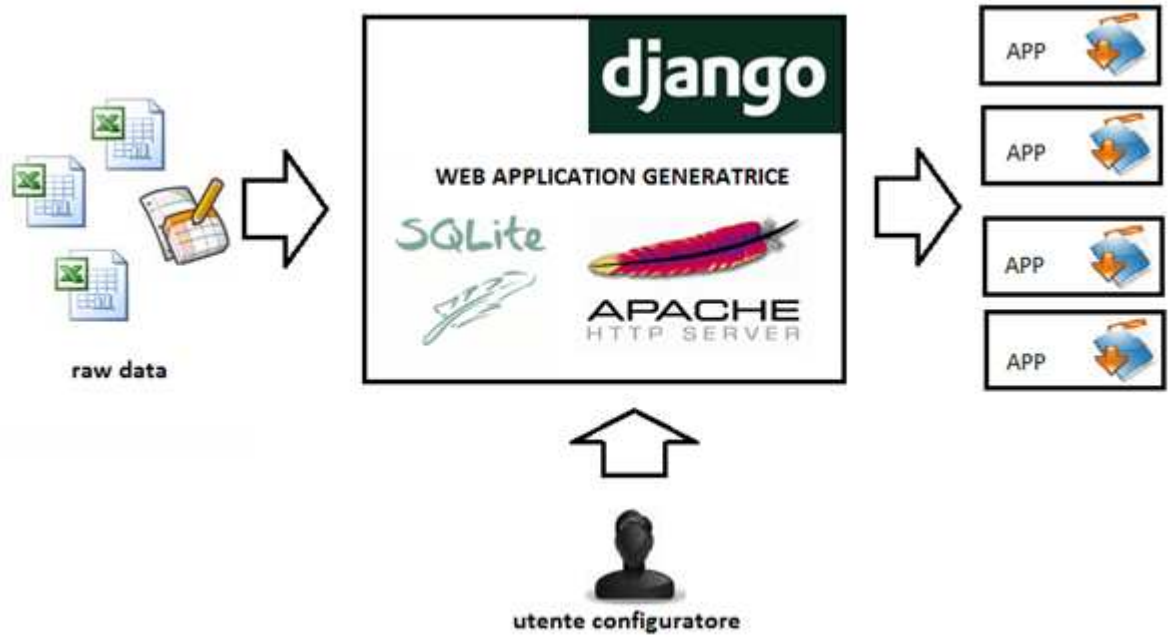

Figura 19 schema applicazione generatrice architettura Django/Apache

Ogni nuova applicazione è rappresentabile come un *catalogo* configurato in un certo modo e associato ad un unico cliente. Dallo stesso file iniziale contenente i dati 'grezzi' sono possibili diverse configurazioni a seconda del numero di colonne presenti.
Tale generazione deve essere efficiente, efficace e soprattutto adatta al maggior numero di casi possibili. Il profilo di questo obiettivo, infatti, per essere compatibile con le linee guida aziendali, deve prevedere un livello di configurabilità e adattabilità per ogni cliente attuale e futuro. In questo modo è possibile gettare le basi per la costruzione di un framework riutilizzabile.

3.2.3 Definizione strutturale web application generatrice

In primo luogo è necessario fissare la struttura dell'applicazione. Questa non deve essere creata da zero in quanto sarebbe un lavoro inutile e dispendioso dal momento che viene utilizzato un framework come django.
Esso, infatti, come descritto nella parte di background permette allo sviluppatore di creare una web application specificandone il modello logico, le viste e mappando le possibili richieste del client.
Nel caso della web application generatrice il modello di dati da sviluppare è quello descritto nella sezione due. Per quanto riguarda le viste invece sono necessarie almeno tre viste differenti. Relative rispettivamente all'inserimento dello spreadsheet, alla configurazione e alla visualizzazione del risultato finale, ovvero alle tabelle del database riempite secondo un certo criterio metodico impostato dall'utente configuratore.
In particolare la prima vista, che da questo punto in poi verrà chiamata *uploader* view, deve consentire all'utente di inviare un file csv o xls presente nella propria macchina, e quindi da locale, o presente nel proprio spazio sul server Google, e quindi da remoto, dopo aver effettuato l'autenticazione.
La seconda vista, la *configurator* view, deve consentire, per ogni attributo presente nel file csv, di specificare come mappare tale attributo nel database creato. Questo procedimento deve permettere all'utente che assume il ruolo di configuratore di scegliere l'entità di destinazione e l'attributo di tale entità da associare. La mappatura effettuata, inoltre, deve essere memorizzabile per essere poi riutilizzata o modificata.
Una volta compiuta la mappatura la nuova web application è stata generata e i dati inseriti non sono

ulteriormente modificabili. I cambiamenti successivi del catalogo vengono gestiti cancellando i dati esistenti e popolando nuovamente la base di dati con il foglio elettronico aggiornato.
Nonostante fosse possibile, infatti, realizzare uno script per la modifica dei dati nel datastore a livello di applicazione web, è stato scelto di seguire le linee guida commerciali.
I dati forniti, e quindi gli aggiornamenti compiuti su essi, vengono sempre rilasciati, infatti, su spreadsheet excel o csv. Questa ipotesi iniziale ha di fatto influito sulla progettazione delle web application relative ai cataloghi. Tuttavia l'implementazione è stata compiuta in modo tale il codice prodotto sia estendibile con i plugin relativi alle funzioni di modifica del db integrati direttamente nel framework e quindi facilmente utilizzabili.
Infine il risultato del processing dei dati è riportato nell'ultima vista creata: la *visualizator* view.
Per ognuna di queste viste deve essere associato, come illustrato nelle sezioni precedenti, l'url corretto nel file urls.py in modo tale possano essere effettuate le richieste e conseguentemente richiamati i metodi sviluppati in views.py.

3.2.4 Base di dati globale

Come descritto nella parte di background django supporta l'object relational mapping.
Questo consente di creare e gestire le entità del datastore come oggetti python mappandone i riferimenti esterni e gli attributi relativi come variabili di esemplare. A questo scopo è necessario definire il modello logico dell'applicazione. Ovvero l'insieme delle classi contenute in models.py.
Per quanto riguarda il progetto sviluppato e discusso in questa tesi viene di seguito riportato il modello implementato(per quanto riguarda la sintassi python si rimanda a [10]).

```
class Base(models.Model):
    deleted=models.NullBooleanField(null= True)
    language=models.CharField(max_length=255,null=True)
    position=models.IntegerField(null=True)
    valid_from=models.DateTimeField(auto_now=False, auto_now_add=False, null=True)
    valid_to=models.DateTimeField(auto_now=False, auto_now_add=False, null=True)
    last_update=models.DateTimeField(auto_now=False, null=True)
    note=models.CharField(max_length=255, null=True)
    def __unicode__(self):
        return self.id
    class Meta:
        abstract=True

class Item(Base):
    code=models.CharField(max_length=255, null=True)
    description=models.CharField(max_length=255, null=True)
    model_code=models.CharField(max_length=255, null=True)
    model_description=models.CharField(max_length=255, null=True)
    def __unicode__(self):
        return self.code

class Outfit(Base):
    code=models.CharField(max_length=255, null=True)
    description=models.CharField(max_length=255, null=True)
    items=models.ManyToManyField(Item, related_name='outfits', null=True)
    def __unicode__(self):
        return self.id

class Item_details(Base):
    sku=models.CharField(max_length=255, null=True)
    bar_code=models.CharField(max_length=255, null=True)
    size_code=models.CharField(max_length=255, null=True)
```

```
    size_description=models.CharField(max_length=255, null=True)
    price=models.FloatField(null=True)
    item=models.ForeignKey(Item, related_name='item_detailss')
    discounts=models.ManyToManyField(Discount, related_name='item_detailss', null=True)
    def __unicode__(self):
        return self.id

class Interface_image(Base): image=models.FileField(null=True,
            upload_to='/Users/Riccardo/Documents/workspace/global_modificato/mysite/data_app/interface/')
    def __unicode__(self):
        return self.id

class Translation(Base):
    translation=models.CharField(max_length=255, null=True)
    translation_code=models.CharField(max_length=255, null=True)
    translation_language=models.CharField(max_length=255, null=True)
    key=models.CharField(max_length=255, null=True)
    def __unicode__(self):
        return self.id

class Attachment(models.Model):
    content_type=models.CharField(max_length=255, null=True)
    entity=models.CharField(max_length=255, null=True)
    entity_id=models.CharField(max_length=255, null=True)
    entity_attribute=models.CharField(max_length=255, null=True)
    type=models.CharField(max_length=255, null=True)
    filename=models.FilePathField(null=True)

    def get_path(self, FileField(null=True, upload_to=get_path)
    def __unicode__(self):
        return self.id

class Pricelist(Base):
    currency=models.CharField filename):
        path=os.path.join('attachment',self.entity)
        return os.path.join(path,filename)

    fullpath=models.
(max_length=255, null=True)
    def __unicode__(self):
        return self.currency

class Discount(Base):
    value=models.CharField(max_length=255, null=True)
    def __unicode__(self):
        return self.value

class Location(Base):
    code=models.CharField(max_length=255, null=True)
    description=models.CharField(max_length=255, null=True)
    pricelist=models.ForeignKey(Pricelist, related_name='pricelists')
    discounts=models.ManyToManyField(Discount, related_name='locations', null=True)
    def __unicode__(self):
        return self.code

class Customer(Base):
    code=models.CharField(max_length=255, null=True)
    description=models.CharField(max_length=255, null=True)
```

```
    name=models.CharField(max_length=255, null=True)
    customer_language=models.CharField(max_length=255, null=True)
    vat_code=models.CharField(max_length=255, null=True)
    locations=models.ManyToManyField(Location, related_name='customers', null=True)
    discounts=models.ManyToManyField(Discount, related_name=’customers’, null=True)
    def __unicode__(self):
        return self.code

class Agent(Base):
    code=models.CharField(max_length=255, null=True)
    description=models.CharField(max_length=255, null=True)
    name=models.CharField(max_length=255, null=True)
    user_name=models.CharField(max_length=255, null=True)
    password=models.CharField(max_length=255, null=True)
    vat_code=models.CharField(max_length=255, null=True)
    locations=models.ManyToManyField(Location,  related_name='agents', null=True)
    discounts=models.ManyToManyField(Discount,  related_name='agents, null=True)
    customers=models.ManyToManyField(Customer ,related_name=’agents’, null=True)
    def __unicode__(self):
        return self.code

class Order(Base):
    code=models.CharField(max_length=255, null=True)
    description=models.CharField(max_length=255, null=True)
    creation_date=models.DateTimeField(auto_now_add=False,null=True)
    customer=models.ForeignKey(Customer, related_name='orders', null=True)
    agent=models.ForeignKey(Agent, related_name='orders', null=True)
    discounts=models.ManyToManyField(Discount, related_name='orders', null=True)
    location=models.ForeignKey(Location, related_name='orders', null=True)
    def __unicode__(self):
        return self.code

class Order_row(Base):
    item_details=models.ForeignKey(Item_details, related_name='order_rows')
    order=models.ForeignKey(Order, related_name='order_rows')
    quantity=models.IntegerField(null=True)
    discounts=models.ManyToManyField(Discount, related_name='order_rows', null=True)
    def __unicode__(self):
        return self.id

class Address(Base):
    customer=models.ForeignKey(Customer, related_name='addresses', null=True)
    mail=models.CharField(max_length=255, null=True)
    phone=models.CharField(max_length=255, null=True)
    zip_code=models.CharField(max_length=255, null=True)
    city=models.CharField(max_length=255, null=True)
    province=models.CharField(max_length=255, null=True)
    address=models.CharField(max_length=255, null=True)
    country=models.CharField(max_length=255, null=True)
    agent=models.ForeignKey(Agent, related_name='addresses', null=True)
    def __unicode__(self):
        return self.id
```

Come è possibile notare la descrizione degli oggetti dato il modello relazionale risulta immediata. La prima classe, *Base*, è sottoclasse di *models.Model*. Tutte le altre classi sono, invece, sottoclassi di Base e quindi anche della classe relativa alla gestione dei modelli.

Sfruttando il paradigma della programmazione ad oggetti questa dichiarazione consente a tutte le classi di utilizzare i metodi e gli attributi di Base e quindi di models.Model.
Per quanto riguarda Base gli attributi non sono dichiarati privati e sono quindi utilizzabili da tutte le sottoclassi. In particolare il concetto di attributi e metodi pubblici e privati in python è molto diverso dai linguaggi C++ e Java, essendo questi ultimi linguaggi compilati. Un attributo dichiarato con l'underscore o il doppio underscore iniziale viene trattato dall'interprete come un attributo privato, altrimenti come pubblico.
Tornando alla classe models.Model, essa contiene i metodi per la creazione, validazione e memorizzazione nel db dell'oggetto e quindi consente di utilizzare il supporto ORM descritto nella parte di background. Ovvero di realizzare e gestire il modello fisico dei dati specificandone il modello logico.
Per quanto riguarda le variabili di esemplare delle classi relative alle entità implementate, esse si suddividono in attributi delle entità del database e riferimenti esterni per le associazioni tra queste.
Ad ogni attributo, a seconda del tipo di dato da memorizzare e gestire, viene associata un'istanza di un oggetto della classe models.
Ogni istanza permette, oltre alla memorizzazione nel database al momento del salvataggio dell'oggetto entità, l'utilizzo di una serie di metodi e argomenti opzionali per l'inserimento, la validazione e la gestione in generale.
La validazione, in particolare, consente di verificare se una data istanza può essere memorizzata come entry del database controllandone quindi i vincoli dichiarativi.
Di seguito viene presentato un elenco dei tipi fondamentali richiesti per gli attributi delle entità descritte nel modello relazionale e le sottoclassi di models con le quali vengono istanziati gli oggetti associati:

- Stringhe: Per quanto riguarda la gestione delle stringhe la classe *models.CharField* consente il salvataggio di un numero variabile di caratteri. Opzionale è l'argomento *CharField.max_length* che permette di specificare la lunghezza massima memorizzabile. L'argomento *models.null* invece viene ereditato da tutte le classi che derivano da models e utilizzato per specificare se l'attributo relativo è opzionale o meno.
- Numeri: *models.IntegerField* consente di memorizzare i numeri interi, mentre *models.FloatField* i numeri decimali.
- Data e ora: In questo caso viene utilizzata la classe *models.DateTimeField*. Questa fornisce come argomenti *DateTimeField.auto_now* e *DateTimeField.auto_now_add*. Il primo consente di memorizzare automaticamente la data e l'ora in diversi formati opzionali, per esempio come *'%anno-%mese-%giorno %ora:%minuti:%secondi'*, al momento della creazione dell'entry, il secondo al seguito di ogni modifica.
  Ovviamente questi due argomenti sono perfettamente compatibili. Nel caso siano inizializzati entrambi a *TRUE* verrà, al momento dell'inserimento dell'entry, memorizzata la data e l'ora attuale.
  Successivamente, in seguito ad ogni modifica, le informazioni vengono aggiornate con i parametri attuali.
- File: Gli attributi che corrispondono a dei file specifici presenti nella memoria di massa vengono mappati nella classe *models.FileField*.
  Per questi attributi la gestione è differente dagli altri in quanto dal catalogo viene fornito il percorso del file, sia esso presente nella macchina locale sia in un server remoto. Da questo percorso l'applicazione deve ricavare tutte le informazioni necessarie per il trasferimento del file nel proprio filesystem. Questo passo è necessario in quanto deve essere possibile compiere un'elaborazione dei file caricati ed inoltre ogni client, come verrà discusso nel corso del capitolo cinque, al momento del download dei dati dal server necessita di scaricare tutti i contenuti presenti nel catalogo.
  La realizzazione dell'upload degli attachment verrà illustrata nella sezione successiva.

Nonostante django supporti relazioni, tra entità, di tipo ‘uno a uno’, ‘uno a molti’ e ‘molti a molti’ sono state utilizzate solo le ultime due, descritte di seguito:

- ‘uno a molti’: Per la realizzazione di questa relazione la classe relativa all’entità corrispondente al lato ‘molti’ deve contenere il riferimento all’unica entry dell’entità corrispondente al lato ‘uno’.
  Tale riferimento viene specificato attraverso l’attributo relativo alla chiave esterna istanza di *models.ForeignKey*.
  Nella base di dati sviluppata un esempio è dato dalla relazione tra ITEM e ITEM_DETAILS, realizzata attraverso l’attributo *item* associato a *models.ForeignKey(Item, related_name='item_detailss')* della classe Item_details. Esso rappresenta la chiave esterna dell’entità ITEM_DETAILS.
- ‘molti a molti’: La relazione ‘molti a molti’ tra due entità viene realizzata in modo molto simile a quello appena descritto. Ovvero attraverso la specifica della variabile di esemplare relativa alla chiave esterna in una qualsiasi delle classi relative alle due entità collegate. La differenza è nel tipo della chiave. In questo caso infatti si tratta di un oggetto *models.ManyToManyFields*.
  Nella base di dati sviluppata un esempio è dato dalla relazione tra ITEM e OUTFIT, realizzata attraverso l’attributo *items=models.ManyToManyField(Item, related_name='outfits', null=True)* della classe Outfit.
  Per ogni oggetto Item è possibile ricavare tutti gli oggetti Outfit associati attraverso l’istruzione *Item.outfits* specificato come *related_name* nella dichiarazione dell’attributo.

Come è possibile notare non compare l’entità TAG. Questo perché è stata sviluppata da terzi una libreria apposita per la gestione dei tag che verrà illustrata nella sezione successiva completa di modello e metodi. In realtà questa libreria è inadatta per i requisiti progettuali richiesti.
È stata infatti sviluppata una sottoclasse di gestione degli attributi dinamici mantenendo la stessa struttura ma modificando il codice secondo le caratteristiche desiderate nella fase di progettazione.

A questo punto il modello è stato riportato e implementato nella struttura django. Tuttavia essendo i dati disponibili nell’azienda relativi al solo catalogo è stato scelto di implementare dal punto di vista della base di dati tutte le entità ma di limitarsi a quelle relative ai dati disponibili per quanto riguarda il caricamento e quindi l’applicazione vera e propria.
Le entità che vengono effettivamente caricate sono Item, Item_details, Outfit, Tag e Attachment.
In primo luogo sembra una drastica riduzione nel numero di entità. L’implementazione invece non assume di mantenere questo insieme inalterato ed è stata svolta in ottica di modularità e quindi di semplicità nell’espansione al modello iniziale. Da citare inoltre il fatto che le entità più complesse da gestire, quali Tag e Attachment, permettono di essere collegate a qualsiasi entità aggiunta in modo semplice ed immediato, come verrà descritto nelle sezioni successive.

3.2.5 Django-tagging

Per quanto riguarda la memorizzazione dei tag, come è stato citato in precedenza, non è stata implementata la classe relativa all’entità direttamente nei modelli.
Questo perché la gestione dei tag, come si evince dal modello relazionale, è differente da quella delle altre entità.
Non sono presenti delle relazioni dirette con alcuna delle altre entità e, a differenza di ATTACHMENT si desidera conservare un’unica entry per tag anche se relativo a diverse altre entry.
Anche se il modello relazionale inizialmente prevedeva la stessa gestione per quanto riguarda i tag e gli allegati, infatti, in fase di sviluppo è stato scelto di lasciare inalterata la scelta per gli attachment ma di riconsiderarla per i tag.
Tale scelta è è stata compiuta in quanto la gestione degli allegati è stata implementata anche nei

database precedenti mentre i tag sono un'innovazione portata dallo sviluppo del database Global e quindi devono essere considerate tutte le possibili soluzioni per poterne scegliere la migliore.
In particolare si desidera, come già citato, che sia presente un unico riferimento per tag, anche se questo è relativo ad un numero variante di altre entry. Essendo i tag l'elemento centrale della base di dati, infatti, è necessario ottimizzarne tutte le operazioni.
Il progetto discusso in questa tesi inoltre è un progetto di ricerca e sviluppo e, come citato precedentemente, devono essere testate, sviluppate e analizzate tutte le soluzioni alternative possibili che possano teoricamente garantire dei buoni risultati.
A tale scopo sono state analizzate due soluzioni possibili:

- Definire l'entità Tag nel modello logico includendo inoltre i metodi necessari per l'inserimento, la cancellazione e la modifica e fornendo un'entità di supporto per la realizzazione della relazione 'molti a molti' con ogni altra entità.
- Sfruttare la libreria *django-tagging*[11] sviluppata da Google come componente additivo per django.

La seconda soluzione rappresenta la scelta logicamente migliore. Nell'ingegneria del software infatti è sempre consigliabile utilizzare moduli di codice già testati, ottimizzati e soprattutto utilizzati da parte del maggior numero di sviluppatori possibili.
È stata però riscontrata una differenza nella definizione concettuale di tag definita all'interno di questo progetto e nella libreria di Google.
Come descritto nel dettaglio nel capitolo due infatti il tag è costituito da un tipo, un valore e un riferimento opzionale ad un altro tag genitore. Nella libreria django-tagging invece viene considerato semplicemente come un singolo valore non tipizzato. Questo comporta la modifica dell'intera struttura base del modello e dei metodi della libreria. È stato comunque scelto di operare in questo modo in quanto la libreria è disponibile open-source ed è quindi possibile modificarla aggiungendo le caratteristiche desiderate.
Il tag viene memorizzato nel database secondo la struttura definita dal framework:

```
class Tag(models.Model):
  name = models.CharField(_('name'), max_length=50, null=True, db_index=True)
  objects = TagManager()


  class Meta:
     verbose_name = _('tag')
     verbose_name_plural = _('tags')


  def __unicode__(self):
     return self.name
```

È possibile notare l'implementazione dell'entità Tag, composta, per quanto riguarda la struttura qualitativa, dal solo attributo *name* in quanto, come già citato, i tag sono considerati delle semplici stringhe non tipizzate e senza alcun riferimento con altre entry della stessa entità.
Oltre all'attributo descritto la classe contiene anche *objects*. Questa variabile di esemplare è un attributo a tutti gli effetti della classe ma non dell'entità. Rappresenta infatti un riferimento ad una classe contenitore di metodi per la creazione, la rimozione e quindi la gestione delle entry dell'entità. Nel dettaglio tale riferimento è indirizzato alla classe *TagManager* che permette di inserire, rimuovere e modificare i tag presenti. Permette inoltre di ottenere la lista degli attributi dinamici associati ad un determinato oggetto, tutti gli oggetti relativi ad una determinata istanza e di operare delle ricerche particolari. Come la lista degli oggetti che hanno in comune una determinata lista di tag, la lista di tutti gli attributi associati ad un determinato modello e quindi a tutte le sue istanze o a un sottoinsieme di queste.
Come è possibile intuire il numeroso insieme di caratteristiche e proprietà fornite rappresenta una delle

motivazioni principali per le qual è stato optato di modificare questa libreria.
La classe *Meta*, definita come classe interna, permette di definire dei meta attributi. Il ruolo di questi ultimi è quello di organizzare le entry relative all'entità impostandone un ordinamento specifico o, come nell'esempio, un nome di riferimento per accedervi. Ovvero per richiamare e visualizzare tutte le istanze della classe definita.
Oltre a Tag e a TagManager sono state implementate le classi TaggedItem e TaggedItemManager,
La classe TaggedItem viene utilizzata per la gestione della relazione 'molti a molti' tra l'entità Tag e qualiasi altra entità.

```
class TaggedItem(models.Model):
  tag          = models.ForeignKey(Tag, verbose_name=_('tag'), related_name='items')
  content_type = models.ForeignKey(ContentType, verbose_name=_('content type'))
  object_id    = models.PositiveIntegerField(_('object id'), db_index=True)
  object       = generic.GenericForeignKey('content_type', 'object_id')

  objects = TaggedItemManager()

  class Meta:
     unique_together = (('tag', 'content_type', 'object_id'),)
     verbose_name = _('tagged item')
     verbose_name_plural = _('tagged items')

  def __unicode__(self):
     return u'%s [%s]' % (self.object, self.tag)
```

Le variabili di esemplare sono rispettivamente:

- *tag*: chiave di riferimento esterna associata al massimo ad un'istanza di Tag.
- *content_type*: chiave di riferimento esterna associata al massimo ad un'istanza di Content_type.
- *object_id*: attributo di tipo intero positivo, ad esso viene assegnato l'id dell'oggetto associato all'entry puntata da tag.
- *object*: riferimento generico, può essere una qualsiasi entry di qualsiasi altra entità ovvero un'istanza di qualsiasi classe definita nel modello logico. Per questo motivo viene istanziata con un oggetto reference di tipo generico.

In questo modo viene creata una associazione 'uno a molti' tra Tag e TaggedItem e una 'molti a uno' tra TaggedItem e qualsiasi altra entità mappata nel modello logico.
Per esempio la creazione del tag 'rosso' applicato all'entry di Item con id=1 avviene nel seguente modo:

```
item=Item(id=1)
item.save()
t=Tag.objects.add_tag(item,'rosso')
```

Viene quindi richiamato il metodo della classe TagManager relativo all'inserimento di un nuovo tag associato all'oggetto item.
Nello specifico vengono svolte le seguenti operazioni: La creazione dell'oggetto contenente come attributo la stringa specificata e successivamente la memorizzazione dei riferimenti all'oggetto appena creato e all'istanza passata come parametro nella chiamata al metodo *add_tag* nell'entità TaggedItem aggiungendo l'info relativa al content type dell'oggetto Item.
La classe ContentType memorizza informazioni specifiche ai modelli dell'applicazione. Per ogni modello viene creata un'istanza composta da:

- *app_label*: nome dell'applicazione
- *model*: nome della classe relativa al modello
- *name*: nome del modello specificato eventualmente dal meta attributo *verbose_name*.

Utilizzandone i metodi è possibile referenziare l'attributo content_type all'istanza di ContentType relativa alla classe Item e quindi definire in generale l'entità al quale si riferisce ogni singolo tag.
Nel caso una singola entry di Tag sia associata a più modelli o a più istanze dello stesso modello verrà comunque memorizzata in modo univoco mentre in TaggedItem saranno presenti più entry con reference all'oggetto e al content type distinti.
Finora è stata data una descrizione dettagliata della struttura di django-tagging senza però specificare come sia stata utilizzata questa classe per la memorizzazione dei tag definiti nell'ambito del progetto in discussione. Come citato precedentemente infatti la libreria nel modo in cui viene fornita è inutilizzabile per le caratteristiche richieste.
È necessario in primo luogo modificare radicalmente la struttura dell'applicazione modificando gli attributi della classe Tag. Per questo motivo sono state aggiunte le variabili di esemplare relative alla tipizzazione e al riferimento con altre istanze della stessa classe.

```
class Tag(models.Model):
  name = models.CharField(_('name'), max_length=50, null=True, db_index=True)
  type = models.CharField(_('type'), max_length=50, null=True, db_index=True)
  parent=models.ForeignKey("self", null=True)
  objects = TagManagerModificato()

  class Meta:
    #ordering = ('name',)
    verbose_name = _('tag')
    verbose_name_plural = _('tags')

  def __unicode__(self):
    return self.name
```

Inoltre è stata implementata una sottoclasse di TagManager: *TagManagerModificato*.
In particolare sono stati ridefiniti attraverso la tecnica dell'overriding alcuni metodi per la gestione dei tag presenti nella superclasse.
Questi metodi infatti, come lo stesso add_tag, non erano stati implementati per gestire i nuovi oggetti definiti. Adottando la logica precedente quello che veniva fatto era semplicemente l'aggiunta della singola stringa ad un'istanza della classe Tag. È stato deciso di effettuare le modifiche proseguendo su questa logica ma aggiungendo il tipo e il possibile genitore. Modificando quindi tutti i metodi presenti aggiungendo le istruzioni per assegnare ai parametri *type* e *parent* i valori corrispondenti al tipo del tag e al genitore.
La creazione di un nuovo oggetto avviene ora nel modo seguente:

```
item=Item(id=1)
item.save()
tag_padre=Tag.objects.add_tag(item, 'colore', 'rosso', None)
tag_figlio=Tag.objects.add_tag(item,'acrilico','90%',tag_padre)
```

Nell'esempio sopra è stato creato un primo tag relativo al colore di un oggetto Item e successivamente un altro relativo alla percentuale di acrilico presente nel colore.
Da notare che questo equivale all'associazione di due ulteriori attributi ad item. Se l'entità fosse stata modellata con questi attributi aggiuntivi l'inserimento sarebbe stato il seguente:

```
item=Item(id=1, colore='rosso', acrilico='90%')
item.save()
```

In primo luogo il metodo descritto sembra più semplice ed immediato. Questo è verificato nel caso tutti gli item siano associati a questi attributi. Nel caso in cui gli item relativi ad un determinato cliente non contengano queste informazioni invece la tabella verrebbe riempita da un numero consistente di entry contenenti attributi inizializzati a None.
Questo rappresenta il contributo fondamentale fornito dall'utilizzo dei tag. è possibile infatti associare dinamicamente gli attributi ad ogni singolo oggetto evitando quindi di doverne fissare la struttura. Struttura che potrebbe non essere adatta per ogni catalogo fornito.

3.2.6 Importing data

L'importazione dei dati relativi al file csv o xls caricato dall'utente nella vista uploader può essere suddivisa in due parti fondamentali: il caricamento delle stringhe e il recupero degli allegati specificati.
Mentre la gestione degli Attachment viene compiuta nella fase di elaborazione dei dati il caricamento dello spreadsheet rappresenta la prima operazione eseguita dall'utente.
Nella uploader view, come citato nelle sezioni precedenti, viene consentito di scegliere tra il caricamento del file da remoto o dal proprio spazio su Google Docs.

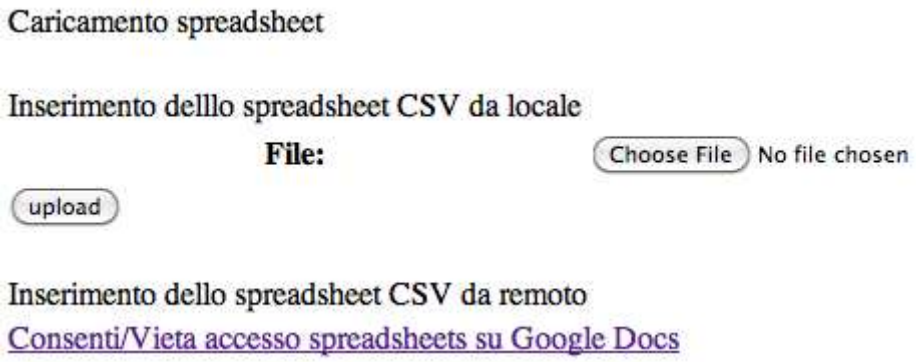

Figura 20 uploader view progetto Django/Apache

Di seguito vengono illustrate le due diverse tecniche sviluppate, le problematiche e i vantaggi introdotti da ciascuna.

3.2.6.1 Importing Data: Google Cloud

Google permette di utilizzare una vasta rete di servizi dal proprio cloud, dalla mail al caricamento di immagini, video e documenti in ambienti di editing professionali che ne consentono la modifica e l'elaborazione.
Nel corso della tesi sono stati analizzati due servizi cloud di Google, uno di cloud computing, Google App Engine(che verrà illustrato nel capitolo quattro) e Google Docs.
Quest'ultimo permette di caricare fino ad 1 GB di spazio virtuale per la memorizzazione di documenti nei formati più utilizzati come .doc, .xls, .csv, ecc..
Essendo il servizio gratuito e offrendo delle risorse simili ai programmi di videoscrittura più utilizzati, attualmente sta diventando la scelta di un numero sempre più ampio di aziende.
Per questo motivo è stato deciso di implementare l'upload dei file relativi al catalogo dal server Google oltre che da remoto.
L'opzione fornita, oltre ad essere dal punto di vista del cliente, e quindi commerciale, valida e conveniente, è stata scelta per il vasto supporto di Google in termini di libreria e documentazione. Per l'utilizzo di tutti i suoi servizi infatti vengono fornite le API necessarie ad ogni tipo di operazione,

ognuna delle quali ben documentata[21].
La release attuale delle API è la 3.0, questa è una versione sperimentale ma, dato che la versione 2.0 non è stata rilasciata per python e la versione 1.0 è deprecata la versione utilizzata è stata quella attualmente in rilascio.
In primo luogo è necessaria l'autenticazione col server Google.
Esistono tre strategie in merito: autenticazione classica inserendo user-name e password, autenticazione AuthSub o autenticazione OAuth.
La prima rappresenta la soluzione più semplice in quanto è sufficiente inserire dall'applicazione le credenziali in termini di nome utente e password per accedere ai documenti presenti:

```
client = gdata.docs.client.DocsClient(source='hUmus-data_app-1.0')
client.ClientLogin('riccardo.lorenzon@gmail.com', 'password', client.source)
feed = client.GetDocList(uri='/feeds/default/private/full/-/spreadsheet')
```

La prima istruzione riguarda la creazione di un client object relativo alla gestione dei documenti. Per autenticare questo oggetto viene richiamato il metodo ClietLogin che, attraverso il nome utente e la password consente di ottenere le credenziali per l'accesso ai documenti. Attraverso queste ultime è possibile accedere a tutti i servizi relativamente all'oggetto che ha effettuato la richiesta.
In questo caso la richiesta è stata fatta da un client object relativo ai documenti quindi il campo d'azione è fissato alla gestione degli stessi.
Nell'esempio fornito viene richiamata l'istruzione *getDocList* che recupera la lista dei documenti allocati nell'uri specificato(si ricorda che uri, *Uniform Resource Identifier*, rappresenta un qualsiasi tipo di risorsa presente nel web) e quindi, in questo caso, la lista di tutti gli spreadsheet.
Nel capitolo successivo, invece, verrà presentato un metodo per gestire i documenti in modo differente, utilizzando un client di tipo *gdata.spreadsheets.client* per recuperare le singole righe e le singole celle contenute nel foglio di lavoro.
Il procedimento illustrato sembra a prima vista semplice ed efficace. Purtroppo però detiene un grande limite: il cliente deve necessariamente fornire il proprio nome utente e password all'applicazione generatrice. Con queste informazioni è possibile utilizzare tutti gli altri servizi ed inoltre i sistemi di sicurezza e di protezione forniti dall'applicazione in fase di sviluppo non sono al livello di quelli dei server Google.
Per questo motivo è stato deciso di effettuare l'autenticazione del cliente utilizzando in modo diretto il server Google attraverso l'interfaccia di autenticazione fornita e, successivamente, utilizzare le credenziali ottenute per effettuare richieste e ottenere i documenti desiderati.
L'applicazione in questo modo non necessita dei dati personali del cliente ma di una stringa fornita dal server con cui effettuare le richieste autenticate. Questa stringa alfa numerica, il *token*, rappresenta il concetto di *ticket* presente in molti sistemi di autenticazione, e serve per garantire al server Google il permesso dell'utente proprietario dell'account accordato all'applicazione che genera le richieste.
La scelta di effettuare l'autenticazione nel modo descritto comporta diversi fattori positivi. Oltre al fatto che il cliente non deve comunicare le proprie credenziali direttamente all'applicazione generatrice è possibile infatti restringere il campo d'azione ad un insieme di azioni fissato.
Sono utilizzabili, a questo scopo, le ultime due strategie citate: autenticazione AuthSub e OAuth.
La prima è una tecnica proprietaria e indicata nel caso in cui il client necessiti di utilizzare applicazioni che necessitano di autenticazione dell'account per essere utilizzate in un server esterno.
La seconda è una valida alternativa a AuthSub in quanto consente anch'essa di ottenere il token di autenticazione senza chiedere direttamente al client le proprie credenziali. Viene consigliata se il client necessita di autenticazioni multiple per ottenere dati da servizi di cloud storage quali per esempio Twitter, YouTube, ecc..
Essendo per il progetto in discussione necessaria la sola autenticazione dell'account Google è stato scelto di utilizzare AuthSub.

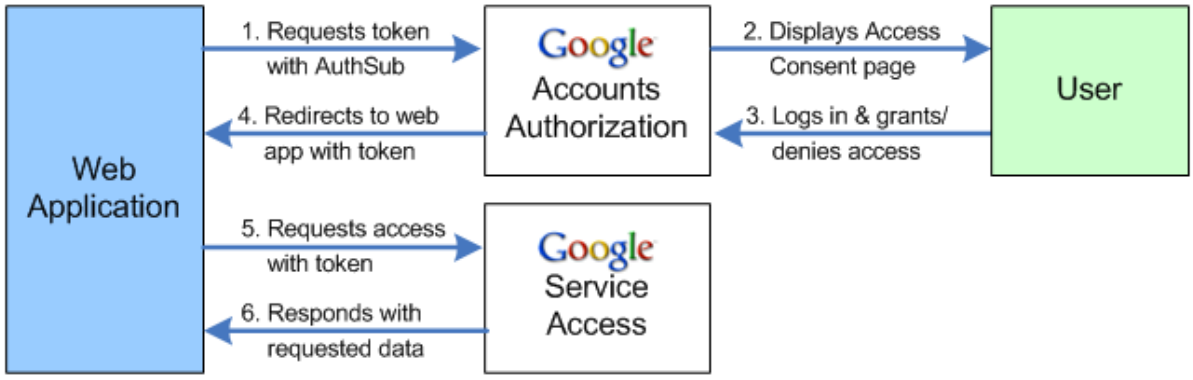

Figura 21 schema AuthSub

Per permettere agli utenti di utilizzare il proprio account Google senza dover fornire le credenziali all'applicazione viene utilizzata, come citato precedentemente, una terza parte: il server di Google relativo all'autenticazione degli account.
L'applicazione, al momento della necessità di accedere ai documenti invia una richiesta per il token di autenticazione al server che risponde con la pagina di richiesta delle credenziali utente. Una volta ottenute le credenziali corrette e confermato il campo d'azione della web application l'utente viene reindirizzato a quest'ultima.
Tutte le richieste successive verranno inviate dall'applicazione generatrice al server di Google relativo ai servizi cloud. Ognuna di queste deve essere provvista del token ottenuto.
Di seguito viene riportato il codice relativo alla richiesta del token da parte dell'applicazione al server di autenticazione degli account:

```
def GetAuthSubUrl():
    next = 'http://127.0.0.1:8000/mysite/index/None/'
    scopes = ['https://docs.google.com/feeds/', 'https://spreadsheets.google.com/feeds/']
    secure = False
    session = True
    return gdata.gauth.generate_auth_sub_url(next, scopes, secure=secure, session=session)
return render_to_response('index.html',{'glink':GetAuthSubUrl(), 'form':form,
'auth':authenticated},RequestContext(request, {}))
```

L'url *glink* permette di accedere alla pagina di autenticazione Google. Al corretto inserimento dei parametri relativi all'utente dell'account viene successivamente chiesta conferma del campo d'azione concesso all'applicazione che ha effettuato la richiesta.
Per quanto riguarda i parametri richiesti per la generazione dell'url:

- *next* specifica l'indirizzo web successivo alla conferma ricevuta dall'utente.
- *scopes* indica il campo d'azione del token che verrà assegnato, a tal proposito il server Google richiederà la conferma all'utente per l'accesso da parte dell'applicazione generatrice ai Documenti e agli Spreadsheet.
- *secure* indica se tutte le richieste inoltrate al server Google contenenti il token di autenticazione concesso devono essere firmate. La firma supportata è la RSA digital signature.
- *session* specifica se il token richiesto può essere promosso a token di sessione. Questa promozione consente di utilizzare lo stesso token un numero illimitato di volte. Nel caso sia impostata a FALSE infatti è consentita una sola operazione per token.

La risposta viene fornita attraverso il metodo render_to_response, che, come è stato descritto precedentemente, consente la visualizzazione dele informazioni contenute nella risposta in un template

html, in questo caso *index.html*.
Inserendo il tag {{glink}} all'interno del codice html esso viene quindi renderizzato e, alla selezione consegue il redirect verso la pagina di autenticazione Google.

Figura 22 Pagina autenticazione Google

Una volta inseriti il nome utente e la password corretti viene visualizzata la pagina relativa all'acquisizione del token. A questo punto viene chiesta conferma all'utente mostrando l'indirizzo IP del server richiedente.

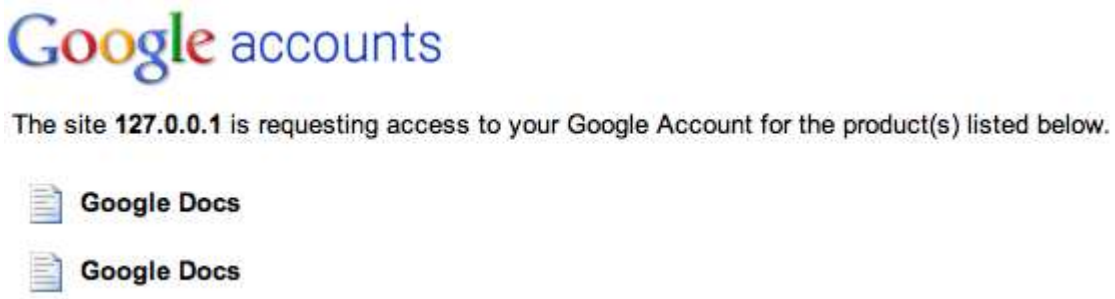

Figura 23 Pagina di richiesta per l'accesso alle risorse Google Cloud

In questo caso lo scope è dato dall'accesso ai documenti e agli spreadsheets e viene segnalato attraverso la lista visualizzata (nell'esempio formata dalle due icone di Google Docs).
Una volta data la conferma il server Google restituisce attraverso il metodo GET il token generato nella forma:

current_url=http://127.0.0.1:8000/mysite/index/None/?auth_sub_scopes=https://docs.google.com/feeds/+https://spreadsheets.google.com/feeds/&**token**=[...]

A questo punto per poter comunicare con il server Google è sufficiente estrarre il token dall'url restituito ed assegnarlo successivamente al client gdocs:

```
gd_client = gdata.docs.client.DocsClient(source='hUmus-data_app-1.0')
single_use_token = gdata.auth.extract_auth_sub_token_from_url(current_url)
gd_client.auth_token=gdata.gauth.AuthSubToken(single_use_token)
```

La prima istruzione è la stessa utilizzata nel caso dell'autenticazione ClientLogin, ovvero viene creato

un client object relativo alla gestione dei documenti. Le istruzioni successive sono per l'estrazione del token e l'autenticazione del client object creato.
Nel caso in cui vengano generate più richieste utilizzando gd_client viene però generata un'eccezione in quanto il token è utilizzabile solo una volta a meno che non venga promosso a *token di sessione*.
A differenza dell'autenticazione discussa precedentemente è possibile notare la necessità di una serie di metodi per la gestione del token.
Per questo motivo viene fornita la libreria gdata.docs.service, contenente i metodi necessari, tra i quali la funzione di promozione:

```
gd_client = gdata.docs.client.DocsClient(source='hUmus-data_app-1.0')
gd_service = gdata.docs.service.DocsService(source='hUmus-data_app-1.0')
single_use_token = gdata.auth.extract_auth_sub_token_from_url(current_url)

gd_service.UpgradeToSessionToken(single_use_token)

gd_client.auth_token=gdata.gauth.AuthSubToken(gd_service.GetAuthSubToken())
feed = gd_client.GetDocList(uri='/feeds/default/private/full/-/spreadsheet')
```

Il codice riportato è relativo all'estrazione e promozione del token utilizzato successivamente per ottenere la lista di spreadsheet presenti.
Per poter utilizzare il token nelle altre viste è necessario salvarlo in una unità di memoria condivisa. Come già citato, infatti, le viste in django non possono condividere puntatori e quindi aree di memoria centrale. Nel progetto illustrato, per quanto riguarda lo sviluppo in questa piattaforma, sono state utilizzate le variabili di sessione.
Queste variabili consentono di memorizzare e recuperare informazioni relative ad ogni utente specifico ogni volta che viene effettuato l'accesso al portale e quindi all'applicazione sviluppata.
L'applicazione generatrice, in questa prima implementazione centralizzata, non è stata progettata per la multi utenza in quanto è stato dato maggiore spessore alla funzionalità ed efficienza nell'elaborazione dei dati.
Le informazioni da registrare nelle variabili di sessione sono quindi relative all'unico utente amministratore, che caricherà il file csv contenente i dati e successivamente configurerà il catalogo.
Nel capitolo successivo verrà invece fornita una prima soluzione multi utente.
Di default il framework django salva le variabili di sessione nel database. Nel salvataggio delle informazioni viene memorizzato inoltre un cookie contenente l'id della sessione relativa all'utente.
Al momento del caricamento dell'applicazione il browser carica i cookie presenti procedendo successivamente a ripristinare le variabili di sessione associate.
Per rendere la procedure più veloce ma meno sicura dal punto di vista del ripristino è possibile scegliere di memorizzare i dati nella cache del browser in modo tale il caricamento sia immediato.
Il problema di questa soluzione risiede nella perdita dei dati conseguente allo svuotamento della stessa struttura di memorizzazione. Per questo motivo come soluzione è stato optato per il primo metodo descritto.
Le variabili vengono gestite attraverso il dizionario *session*, attributo di HttpRequest. In questo modo ogni vista ha la possibilità di recuperare il token ed effettuare richieste al server Google.
Successivamente all'estrazione e alla promozione del token esso viene memorizzato in una variabile di sessione globale a tutte le richieste: *session['token']*.

```
request.session['token']=gd_service.current_token
```

Il ripristino delle credenziali nelle altre viste avviene recuperando il valore dalla variabili e assegnandolo a gd_service:

```
gd_service.current_token=request.session['token']
```

La scelta fatta in tal senso risulta efficace. In realtà sono possibili altre soluzioni. Una di queste era la gestione manuale nel database di una tabella Token attraverso un modello django. Tuttavia la memorizzazione utilizzando le variabili di sessione per un dizionario quale *session['token']* risulta immediata come è possibile notare dall'esempio. Mentre per l'utilizzo manuale del database è necessario serializzare l'oggetto *gd_service.current_token* in un insieme di dati adatti al salvataggio nella base di dati.
Una volta ripristinato il token deve essere caricata nella pagina di uploading la lista degli spreadsheet contenuti nello spazio Google dell'utente.
A questo proposito viene richiamata la funzione per ottenere la lista degli spreadsheet presenti citata precedentemente. L'output prodotto da questa viene renderizzato nella pagina html.

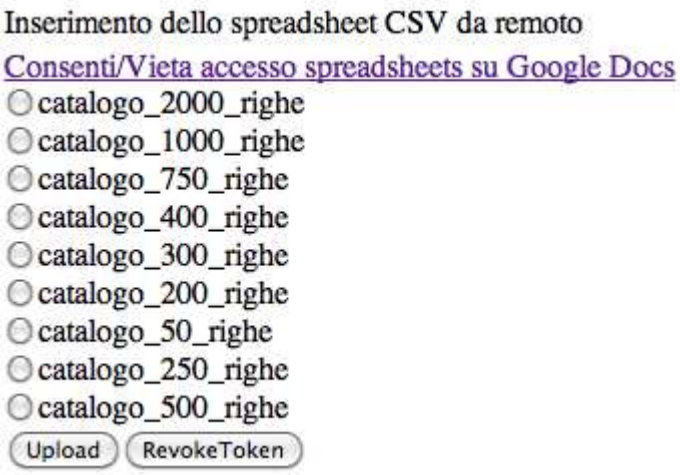

Figura 24 Form iniziale con lista spreadsheet caricata da Google Cloud

Scelto lo spreadsheet da caricare nel server, attraverso la dichiarazione di un'istanza di HttpResponseRedirect, l'esecuzione passa nella vista di configurazione.

3.2.6.2 Importing data: client remoto

Dopo aver descritto la parte relativa al caricamento del file contenente i dati 'grezzi' dal server Google viene ora presentata la soluzione implementata per il caricamento da client remoto.
In fase di progettazione infatti è stato scelto di permettere all'utente di caricare fogli di lavoro presenti nella propria macchina.
A questo scopo nella pagina iniziale è stata inserita una form relativa al caricamento del file csv da locale.
Una form django è una sottoclasse di *forms.Form* i cui attributi corrispondono alle informazioni da compilare nel modulo dell'interfaccia web. In questo caso è necessaria solo l'indicazione della locazione del file in memoria di massa. Viene quindi instanziato un attributo di tipo *forms.FileField*.
Come è possibile notare la struttura della form è molto simile a quella dei modelli descritta nella sezione relativa alla base di dati. In realtà la struttura di definizione degli attributi è la stessa, siano essi relativi ad un modello che ad una form. La differenza è insita nella gestione degli stessi. In questo caso infatti vengono utilizzati per raccogliere informazioni dall'interfaccia utente.
La classe definita è la seguente:

```
class UploadFileForm(forms.Form):
        file=forms.FileField(required=False)
```

In modo equivalente è possibile definire la form come segue:

```
from django.forms import ModelForm
class UploadFile(models.Model):
        file=models.FileField(required=False)
class UploadFileForm(ModelForm):
        class Meta:
                model=UploadFile
```

La definizione della classe Meta annidata all'interno della classe UploadFileForm consente di definire il riferimento al modello relativo alla form creata. Questo esempio accentua il concetto citato precedentemente, ovvero l'equivalenza tra le dichiarazioni dei modelli e delle form.
Una volta creata la form deve essere renderizzata nell'interfaccia utente fornita. A questo scopo viene utilizzato nuovamente il metodo render_to_response inserendo nel corpo della risposta un'istanza di UploadFileForm.

```
render_to_response('index.html',{'glink':GetAuthSubUrl(), 'form':form},RequestContext(request, {}))
```

E in index.html la form viene gestita nel modo seguente:

```
<form action="http://127.0.0.1:8000/mysite/index/Upload/" enctype="multipart/form-data" method="POST">
        <td>Inserimento delllo spreadsheet CSV da locale</td><td>{{ form }}</td>
        <td><input type="submit" value="upload"></td>
    </form>
```

Al corretto inserimento da parte dell'utente e al submit dell'informazione fornita viene generata una richiesta al server 127.0.0.1:8000 dell'url mysite/index/Upload/.
A questo punto viene effettuato il controllo della presenza di tale indirizzo e il reindirizzamento alla vista specificata, ovvero alla configuration view.

3.2.7 Configurazione

3.2.7.1 Configurazione: Form dinamica

Dall'upload effettuato il foglio dati viene importato e salvato in una cartella locale del server. A questo punto la provenienza dal server Google o dal client remoto è nulla in quanto qualsiasi modalità sia stata utilizzata il risultato è lo stesso.
Finora non è mai stata discussa la struttura di questo file. É necessario per procedere alla spiegazione della configurazione specificare come siano organizzati i dati riportati nel foglio elettronico.
Sia che lo spreadsheet sia di tipo csv o xls la rappresentazione è invariante. Ovvero un insieme di righe ciascuna contenente un insieme di celle. Ogni colonna inoltre rappresenta un gruppo di dati appartenenti ad una classe il cui riferimento per la rappresentazione viene riportato nella prima riga.
È stato inserito, nell'esempio sottostante, un semplicissimo catalogo csv:

code	description	size	colore	sku	
GPelle 1	giaccone di pelle	1	l	marrone	1
GPelle 2	giaccone di pelle	2	xl	nero	2
GPelle 2	giaccone di pelle	2	48	nero	3
GPelle 3	giaccone di pelle	3	m	grigio	4

Ovviamente l'applicazione generatrice deve essere in grado di gestire dei fogli di calcolo molto più complessi: composti da migliaia di righe e decine di attributi tipizzati in tutti i modi possibili.
Per la configurazione è necessario specificare la destinazione di ogni attributo nella base di dati Global, ovvero quella sviluppata nella sezione due.
Ogni colonna specifica un attributo che deve essere mappato in una entità creando se necessario i

reference alle altre attraverso attribuzione delle chiavi esterne relative.
Per ogni attributo viene chiesto all'utente che assume il ruolo configuratore di scegliere:

- entità di destinazione.
- attributo dell'entità di destinazione.
- parent, nel caso il mapping venga effettuato verso l'entità Tag e sia necessario specificarne il genitore.
- primary key, scelta booleana. Se TRUE allora tale attributo fa parte della chiave primaria dell'entità e non devono essere quindi presenti delle copie duplicate.
- regex, campo adibito all'inserimento di una espressione regolare. Se un qualsiasi valore contenuto nella colonna relativa non è compatibile con l'espressione fornita viene trasformato nella stringa 'not validated!'.

Per realizzare il modulo di raccolta di questi dati sono sorte una serie di problematiche.
In primo luogo il numero degli attributi(e quindi delle colonne) del file csv non è fissato. Questo comporta all'impossibilità di definire la form a priori. È quindi necessario specificarne gli attributi in tempo reale nel momento in cui viene letto il contenuto del file inviato dall'utente.
Il linguaggio di programmazione utilizzato in questo senso fornisce un'ottima soluzione: permette di definire dinamicamente il nome ed il tipo degli attributi di una generica form.
Per la creazione dinamica è sufficiente modificare il metodo costruttore della classe e instanziare un nuovo attributo per colonna del foglio elettronico semplicemente aggiungendo un nuovo elemento alla variabile di esemplare ereditata dalla superclasse forms.Form: *fields*.
Il codice relativo alla classe del modulo di configurazione viene riportato di seguito:

```
class UploadDynamicForm(forms.Form):
    def __init__(self,*args,**kwargs):
        super(UploadDynamicForm,self).__init__(*args,**kwargs)
        for x in first_csv_row:
          self.fields[x]=forms.ChoiceField(choices=[...])
          self.fields["mapping_for_"+x]=forms.ChoiceField(choices=[...])

          def scelte():
              yield('None','None')
              for x in first_csv_row:
                 yield(x,x)

          self.fields["parent_for_"+x]=forms.ChoiceField(scelte())
          self.fields["primary_key_"+x]=forms.ChoiceField(choices=[("False","False"),("True","True")])
          self.fields["regex_for_"+x]=forms.CharField(required=False)
```

UploadDynamicForm, sottoclasse di forms.Form, rappresenta la classe relativa alla form di configurazione. Di questa classe è stato implementato il solo metodo costruttore __init__().
La prima istruzione richiama il costruttore della superclasse. I due parametri contenuti nella chiamata al metodo vengono utilizzati in modo tale possa essere definito un numero qualsiasi di attributi, per questo motivo vengono chiamati *keyword arguments*.
Il primo parametro ***args* crea una tupla comprendente tutti gli attributi passati nella chiamata definiti singolarmente. ****kwargs* invece crea un dizionario comprendente tutti i valori passati nella chiamata contenenti '='.
La variabile *first_csv_row* rappresenta una lista contenente i valori delle celle della prima riga del foglio dati. Nell'esempio riportato precedentemente assume il valore:

```
first_row_csv=[code, description, size, colore, sku]
```

Essa rappresenta quindi l'insieme delle classi di appartenenza dei dati di ogni riga dello spreadsheet. Fortunatamente python permette di aprire il file caricato nel server come file csv e di accedervi riga per riga o cella per cella. Questo rende semplice instanziare first_row_csv e accedervi ai dati essendo una lista di elementi iterabile.
Il ciclo successivo assegna una serie di variabili alla form ad ogni iterazione. Il numero di iterazioni è pari al numero di attributi del file csv.
Le prime istruzioni del ciclo creano, ad ogni iterazione, le seguenti variabili:

- ***x***: entità di destinazione dell'attributo
- *mapping_for_**x***: attributo dell'entità di destinazione
- *parent_for_**x***: istanza associata(valida solo per Tag)
- *primary_key_**x*** : indica se l'attributo fa parte della chiave primaria dell'entità di destinazione
- *regex_for_**x*****:** espressione regolare della quale è necessario verificare la compatibilità con la stringhe del campo associato.

La variabile ***x*** rappresenta il nome del campo dello spreadsheet relativo all'iterazione corrente. Successivamente a ciascuna viene assegnato un oggetto ChoiceField per la tipizzazione. In questo modo la rappresentazione nella form di ognuna di queste variabili è data da una combo box contenente i dati di *choices*.
Particolare rilevante la presenza di un metodo *scelte()* il cui output viene assegnato alla combo box relativa al parent dell'attributo. Come già citato infatti il campo parent viene utilizzato solo quando il mapping avviene verso l'entità Tag. Essendo questa l'unica ad avere una relazione ricorsiva con sè stessa.
Per quanto riguarda i tag non viene permesso all'utente di cambiare il nome dell'attributo.
Questo implica che il tipo di ogni attributo dinamico presente corrisponde al campo associato contenuto nello spreadsheet.
La scelta del parent quindi deve ricadere in uno di questi, motivo per cui choices di parent_for_x contiene le coppie ('attributo', 'attributo') ognuna delle quali è data da un elemento di first_csv_row duplicato. L'istruzione *yield* permette a tale scopo di rilasciare l'output in modo additivo senza dover necessariamente costituire l'istruzione di ritorno dalla procedura.
L'ultimo campo, di tipo *forms.CharField*, viene visualizzato come una casella di testo generica e utilizzato per l'inserimento di espressioni regolari.
Di seguito è stato riportato uno screenshot relativo al mapping di una colonna generica del foglio elettronico:

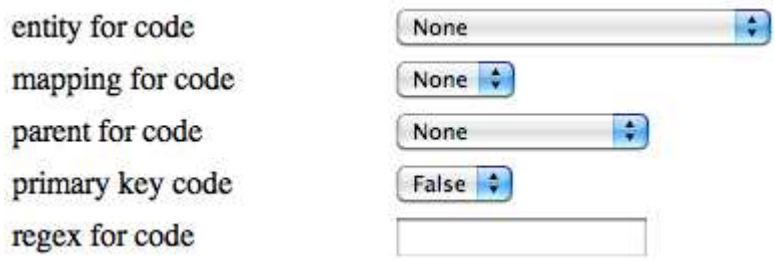

Figura 25 Set di attributi della form di configurazione relativi al campo 'code' del file selezionato

Una volta compilata la form i dati vengono inviati al server in modo tale possano essere elaborati. L'invio avviene utilizzando il metodo POST. Ogni variabile della form, associata con il valore assegnato dall'utente, è contenuta nel dizionario request.POST.

3.2.7.2 Configurazione: Rendering user-friendly

Per l'interfaccia utente relativa alla configurazione è necessario, nel corso della progettazione, tenere

conto del fattore grafico. Solitamente nei progetti ingegneristici questo problema non viene preso in considerazione. In questo caso però è strettamente legato alla semplicità d'uso dell'applicazione generatrice.
In primo luogo infatti i campi della form da compilare erano stati impostati come oggetti forms.CharField.
È stato però riscontrata la difficoltà nella compilazione di questa legata al numero di attributi e delle entità da mappare. Per questo motivo è stato scelto di utilizzare delle combo box in modo tale le scelte fossero vincolate a certi valori.
A questo punto è sorto un altro problema: la scelta di una determinata entità, per il mapping di una specifica colonna del foglio elettronico, implica l'esistenza di vincoli associati all'insieme possibile di attributi selezionabili.
Per esempio si consideri la colonna 'colore' del file csv riportato precedentemente. La scelta di mappare questo attributo nell'entità Item non permette di associarlo per esempio all'attributo *Item_details.sku* o *Outfit.code* in quanto appartenenti ad altre entità.
La scelta dell'entità quindi deve precludere quella di attributi non appartenenti ad essa.
A questo proposito è stato sviluppato uno script tale che, al momento della scelta dell'entità la combo box relativa al mapping del campo specificato venisse riempita correttamente.
Per prima cosa l'attributo choices di ogni combo box definita nel costruttore della form viene riempito con qualsiasi scelta possibile effettuabile su tale modulo.
Successivamente ogni attributo della form differente da regex_**x** viene renderizzato instanziando un tag html di tipo *select* interno alla form con lo stesso nome.

```
<tr>
        <td><font>{{field.label}} </font></td>
        <td>
                <select name="{{field.name}}" id="id_{{field.name}}">
                <option value="{{value}}">{{value}}</option>
                </select>
        </td>
</tr>
```

Mentre nella dichiarazione della combo box relativa alle entità deve essere indicato l'evento da gestire e lo script da richiamare. In questo caso l'evento è la selezione di un elemento mentre lo script implementato da richiamare è quello relativo al riempimento delle altre combo box. Devono essere infatti inseriti gli attributi compatibili ed inoltre, se l'entità non è Tag, deve essere azzerato il numero di elementi in parent.

```
<select name="{{field.name}}" id="id_{{field.name}}" onchange = "setOptions('{{field.name}}',
this.options[this.selectedIndex].value)">
```

Lo script richiamato, *setOptions*, accetta in input il nome della combo box che ha generato la richiesta e il valore selezionato. Il primo parametro è necessario per risalire al modulo da modificare. Il campo ***x*** della form da cui parte la richiesta infatti permette di risalire al campo contenente gli attributi mappabili *mapping_for_**x*** e al campo relativo al parent *parent_for_**x***.
Successivamente all'identificazione l'inserimento di elementi nel select relativo è semplice e immediato.

3.2.7.3 Configurazione: serializzazione

Uno dei concetti fondamentali espressi nel corso di questo progetto riguarda la riutilizzabilità delle configurazioni effettuate in modo tale l'applicazione generatrice possa essere autonoma.
A questo scopo la form compilata dall'utente deve essere in primo luogo scaricabile dal client. Per

essere successivamente riutilizzata deve essere invece implementato un metodo specifico per inviarla al server. Il processo da implementare è chiamato *serializzazione* della form. In altre parole un oggetto complesso quale il modulo strutturato deve essere convertito in un insieme di dati lineare per essere spedito dal server al client attraverso la rete.
Inoltre questa conversione non deve alterare le proprietà quali i riferimenti tra le etichette relative ai campi e i valori con i quali questi sono stati compilati. La serializzazione compiuta deve essere, infatti, un processo totalmente reversibile.
La soluzione scelta consiste nello scrivere su un file tutte le info contenute nell'oggetto complesso sottoforma di stringhe strutturate secondo un certo protocollo. Al momento dell'upload da parte del client il server avrà modo di utilizzare una struttura solida dal quale estrarre tutte le informazioni rendendo quindi il processo reversibile.
Riguardo all'implementazione, django offre come soluzione l'esportazione in file xml o json. Essendo quest'ultimo lo standard maggiormente utilizzato nell'azienda in cui è stato sviluppato il progetto è stato adottato come riferimento. Tale scelta è stata effettuata, inoltre, per l'utilizzo che verrà compiuto nello sviluppo del client IPad(che verrà illustrato nel capitolo quinto).
La libreria disponibile nel framework per la gestione dello standard json è *simplejson*. Essa fornisce i metodi per la conversione di una variabile dizionario in un file json o una stringa contenente la codifica strutturata. Inoltre essendo, come citato precedentemente, un processo necessariamente reversibile vengono fornite le funzioni per ricavare il dizionario partendo da un file o una stringa contenente codice strutturato secondo lo standard json.
Nel progetto la conversione è stata attuata tra la variabile corrispondente al modulo compilato, e quindi ai campi indicizzati e le scelte effettuate dall'utente, e il file json corrispondente, avviando automaticamente il download di quest'ultimo.
Nello specifico è stato creato nel filesystem del server un file json vuoto. In esso è stato scaricato il contenuto del modulo passando come parametro request.POST, che si ricorda è una variabile dizionario contenente i dati passati al server al momento della richiesta da parte del client, attraverso l'istruzione *simplejson.dump*.
In questo modo è possibile trasferire in blocco tutte le informazioni di configurazione.
Successivamente, per avviare automaticamente il download del file è sufficiente utilizzare l'oggetto HttpResponse inserendo come argomento il file stesso e la stringa descrittrice del tipo.
Successivamente per specificare che si tratta di un allegato vengono associati alla chiave *Content-Disposition* dell'oggetto relativo alla risposta i parametri associati al tipo di allegato e al nome con il quale viene scaricato nel filesystem del client.

```
response=open(json_path,mode="w")
simplejson.dump(request.POST, response, ensure_ascii=False, encoding='utf-8')
response=HttpResponse(file, mimetype='application/json')
response['Content-Disposition']='attachment; filename= stream_json.json'
```

Per quanto riguarda invece l'operazione inversa, nella configurator view compare l'indicazione per l'upload di file atti al riempimento automatico del modulo di mapping.
L'implementazione viene eseguita utilizzando la funzione *simplejson.load*. Essa permette di inserire in un dizionario i dati contenuti nel file json.
Nel progetto, confermato il json da utilizzare, il contenuto di esso viene scaricato in una variabile dizionario. Successivamente la variabile inizializzata viene passata assieme alle altre al template utilizzato per la configurazione dinamica. In questo modo al caricamento della pagina il browser visualizzerà i dati passati dal server relativi al json caricato.
Per la visualizzazione e l'inserimento dei dati nel template viene utilizzato il metodo render_to_response utilizzando i tag html forniti da django.

3.2.8 Processing dei dati

Le sezioni precedenti sono state utilizzate per la descrizione dell'import dello spreadsheet e della configurazione effettuata. Il passo successivo, che costituisce il core dell'applicazione generatrice, è dato dall'elaborazione compiuta per ottenere le tabelle del database riempite nel modo specificato e quindi la web application relativa al catalogo.
A priori sono constatabili diverse problematiche. In primo luogo gli attributi devono essere distribuiti su tabelle distinte ma non indipendenti. Deve essere quindi implementato un metodo per l'assegnamento dei riferimenti e il controllo della consistenza.
Inoltre la caratteristica principale di ogni entry è data dal numero di tag ai quali è collegata. Essendo questi ultimi gestiti diversamente dagli altri oggetti, come è stato illustrato nelle sezioni precedenti, anche il loro caricamento deve essere effettuato in modo diverso.
Infine, per quanto riguarda gli allegati, essi si distinguono come i tag dalle altre componenti del modello logico e sono gestiti in modo totalmente diverso. Per questi oggetti, inoltre, non è sufficiente la memorizzazione nel datastore ma è necessaria l'importazione nel filesystem del server del contenuto indicizzato.
L'elaborazione compiuta riguarda, come citato precedentemente, il riempimento del database con l'insieme di dati 'grezzi' fornito in input osservando le specifiche date dall'utente nella configurazione.
Il primo metodo richiamato implementa l'inizializzazione delle variabili. Ovvero inserisce il contenuto del modulo di configurazione, derivato dalla richiesta inviata al server, in una struttura che ne semplifichi la gestione.
Il dizionario request.POST, infatti, contiene tutte le informazioni specificate nella form utilizzando come chiave l'etichetta del parametro mentre come valore il contenuto inserito dall'utente.
Per semplicità nel corso della spiegazione il nome *attributi* è riferito alle variabili di esemplare dei modelli, *campi* ad ogni classe di rappresentanza dello spreadsheet e *parametri* alle variabili di esemplare della form.
Il dizionario ottenuto dalla configurazione non fornisce alcun collegamento immediato tra i parametri. In questo modo dal campo inserito in una certa entità non è possibile risalire direttamente al nome dell'attributo con il quale viene mappato, all'espressione regolare o agli altri attributi presenti.
Per risolvere questo problema è stata utilizzata la caratteristica di univocità relativa ai nomi dei parametri del modulo. Conoscendo il nome del campo **x** è possibile ricavare tutte le altre informazioni estraendole dal dizionario aggiungendo i prefissi *mapping_for_**x***, *parent_for_**x***, *primary_key_**x*** e *regex_for_**x***.
Utilizzando questa osservazione è possibile ricavare, per ogni entità presente, le informazioni su tutti i campi che devono essere mappati come attributi.
Per la verifica dei campi che sono stati mappati inizialmente è stato pensato di controllare tutte le chiavi presenti nel dizionario. Essendo però il loro numero pari a cinque volte quello dei campi tale decisione è stata modificata. Viene, infatti, ricavata nuovamente la prima riga dello spreadsheet e, per ogni elemento di questa ne viene controllata l'esistenza. Se la verifica va a buon fine nel dizionario relativo all'entità in cui viene mappato il campo viene inserito la coppia <attributo mappato, indice della colonna del campo>.
Viene fornito di seguito il codice per la memorizzazione della configurazione effettuata per un campo relativo all'entità Item:

```
if dictionary[key]=='ITEM':
        item[dictionary["mapping_for_"+key]]=first_row_csv.index(key)
        trigger['item']=True
        if dictionary["primary_key_"+key]=='True':
           primary_key_item=primary_key_item+[dictionary["mapping_for_"+key]]
        regex_item[dictionary["mapping_for_"+key]]=dictionary["regex_for_"+key]
```

Il motivo per il quale viene mantenuta traccia dell'indice di colonna piuttosto che il nome del campo stesso è dato dal fatto che, per accedere al contenuto classificato nelle righe dello spreadsheet, il nome del campo costituisce una chiave utile ma il procedimento per ricavare il valore associato è sensibilmente lento. Salvando direttamente l'indice invece, la procedura diventa veloce ed immediata.
In realtà è possibile mantenere tutte le informazioni nell'unico dizionario iniziale. Questo rende sicuramente più semplice l'implementazione dal punto di vista della quantità di codice ma, considerando che uno dei requisiti principali è l'efficacia dell'applicazione generatrice, è necessario ottimizzarne i processi.
In quest'ottica quindi è stato deciso di utilizzare un dizionario distinto per modello, chiave primaria ed espressione regolare. Inoltre sono stati predisposti degli *attivatori*(trigger) per segnalare il mapping di almeno un campo per l'entità specifica. In questo modo non è necessario controllare la presenza di configurazioni specifiche su tutti i modelli presenti. Nel caso in cui nessun campo venga mappato in una determinata entità, infatti, l'attivatore non viene inizializzato.
Per quanto riguarda gli altri parametri, se la chiave primaria relativa a un determinato campo è stata impostata a TRUE l'attributo viene aggiunto alla lista degli elementi relativi alla chiave primaria di quell'entità specifica. L'espressione regolare specificata, infine, viene memorizzata nel dizionario relativo associandola all'entità e all'attributo ricavati dalla form.
La configurazione di campi in Attachment comporta un'ulteriore problematica: ogni singola riga dello spreadsheet può contenere più di un allegato, sia esso relativo agli Item, agli Item_details o agli Outfit. Non è quindi possibile utilizzare un unico oggetto relativo alla memorizzazione.
La necessità di mappare lo stesso oggetto in diverse entry corrisponde, nelle entità differenti da Tag e Attachment, alla presenza di un numero associato di righe nello spreadsheet. Per esempio, se un oggetto Item compare in più Outfit non viene specificato nella stessa riga del foglio elettronico. Vengono poste infatti un numero di righe pari al numero di Outfit in cui compare il capo d'abbigliamento.
Per quanto riguarda Attachment, invece, è necessario creare una struttura che consenta la memorizzazione di numerosi attachment contenuti nella stessa riga. Di seguito viene riportato il codice per la memorizzazione delle informazione relative alla configurazione di un campo mappato come Attachment:

```
if dictionary[key]=='ATTACHMENT_ITEM':
        count['attachment_item']=count['attachment_item']+1
        attachment_item['entity_attribute%d'%count['attachment_item']]=key
        attachment_item['fullpath%d'%count['attachment_item']]=first_row_csv.index(key)
        trigger['attachment_item']=True
        regex_attachment_item['%d'%count['attachment_item']]=dictionary["regex_for_"+key]
```

A tutte le chiavi del dizionario che devono essere inserite nella stessa entry della tabella Attachment viene associato un contatore corrispondente al numero relativo nella lista completa.
Se fosse stato utilizzato il metodo relativo agli altri modelli, infatti, ogni attributo per il quale viene configurato il mapping verso un allegato andrebbe a sovrascrivere quello precedente.
Per ogni attributo di entità diverse da Attachment e Tag, come già citato, è consentito al massimo un campo associato. Tutti i campi configurati successivamente relativi alla stessa entità vanno semplicemente a sovrascrivere il precedente.

I tag vengono memorizzati seguendo un approccio simile a quello utilizzato per gli Attachment. Ovvero essendo molti gli attributi che, secondo le analisi compiute nei dati disponibili, dovranno essere mappati come tali, deve essere considerata, come per gli allegati, la possibilità di doverne mappare più di uno per riga.
Anche in questo caso infatti è stato utilizzato un contatore specifico per distinguere e memorizzare tutti gli attributi mappati come tag.

Inoltre la presenza del parent implica l'associazione con il tag memorizzato con un altro specificato come genitore. Per questo motivo vengono memorizzate nella stessa struttura dizionario con lo stesso contatore le informazioni relative *parent_type* e *parent_name*.
Come per Attachment viene riportato il codice per la memorizzazione dei dati relativi alla configurazione di un campo mappato come attributo dinamico:

```
if dictionary[key]=='TAG_ITEM':
        count['tag_item']=count['tag_item']+1
        tag_item['type%d'%count['tag_item']]=key
        tag_item['name%d'%count['tag_item']]=first_row_csv.index(key)
        if (dictionary["parent_for_"+key]!='None'):
                tag_item['parent_type%d'%count['tag_item']]=dictionary['parent_for_'+key]

                tag_item['parent_name%d'%count['tag_item']]=first_row_csv.index(dictionary['parent_for_'+key])
        else:
                tag_item['parent_type%d'%count['tag_item']]=''
                tag_item['parent_name%d'%count['tag_item']]=default_index
        trigger['tag_item']=True
        regex_tag_item['%d'%count['tag_item']]=dictionary["regex_for_"+key]
```

Una volta compiuta l'inizializzazione viene avviato il ciclo di fetching delle righe dello spreadsheet e inserimento dei dati estratti da queste nelle tabelle del database. Questa procedura consiste nel corpo dell'applicazione generatrice. Le prestazioni di questo ciclo andranno quindi ad incidere sensibilmente in quelle finali.
L'inserimento nel database viene effettuato per tutte le righe contenute nello spreadsheet, viene quindi istanziato un ciclo for che, per ognuna di queste, ne salvi il contenuto in una lista aggiungendo alla fine l'elemento di default: un puntatore a *None*. Lo scopo dell'assegnazione compiuta consiste nell'impostare un indice di default al quale venga associato il valore degli attributi che non sono stati mappati.
Successivamente, per ogni entità, viene controllato il valore dell'attivatore relativo. Solo se quest'ultimo vale TRUE infatti vengono valutati i parametri di configurazione relativi in quanto altrimenti nessun parametro sarebbe stato mappato e ogni dizionario relativo all'entità sarebbe vuoto.
Viene in primo luogo controllata, per ogni entità attivata, la consistenza della chiave primaria.
Per il controllo viene sfruttata la funzione di filtro sulle entry del database fornita da django.
Attraverso la funzione *filter* è possibile, per ogni attributo che fa parte della chiave, controllare la presenza di oggetti con l'attributo specifico istanziato allo stesso valore. In questo modo è possibile recuperare l'oggetto con la chiave duplicata.
Viene di seguito riportato il codice per il controllo della consistenza della chiave primaria per l'entità Item:

```
if primary_key_item.__len__()!=0:
            queryset=Item.objects.all()
              for i in range(0,primary_key_item.__len__()):
                 if primary_key_item[i]=='code':
                    queryset=queryset.filter(code=x[item['code']])
                 if primary_key_item[i]=='description':
                    queryset=queryset.filter(description=x[item['description']])
                 if primary_key_item[i]=='model_code':
                    queryset=queryset.filter(model_code=x[item['model_code']])
                 if primary_key_item[i]=='model_description':
                    queryset=queryset.filter(model_description=x[item['model_description']])
```

Se riscontrata la presenza di una duplicazione di chiave viene segnalato su un file di log l'errore riscontrato. Altrimenti l'esecuzione procede con l'inserimento dell'entry relativa nella tabella. Questa

operazione, grazie al supporto ORM fornito da django, non deve necessariamente essere realizzata utilizzando la sintassi SQL.
È sufficiente infatti utilizzare il metodo ereditato da tutti le sottoclassi di models.Model: *get_or_create*:

```
objects.get_or_create(lista_attributi_validati)
```

È necessario specificare, come argomenti del metodo, gli attributi relativi al modello e il valore assegnato. Questa istruzione effettua in primo luogo una ricerca di eventuali entry con lo stesso set di attributi specificato. L'inserimento viene effettuato solo se non viene riscontrata la presenza di nessuna entry di questo tipo.
Ogni attributo inoltre non viene inserito direttamente come argomento della funzione ma ne deve essere prima controllata la consistenza con l'eventuale espressione regolare fornita dal configuratore. È stata quindi creata una funzione che riceve in input l'espressione regolare relativa al campo mappato e il valore da mappare. Se i due elementi sono compatibili l'inserimento avviene esattamente come descritto precedentemente, altrimenti viene segnalato l'errore sul file di log e restituita la stringa 'not validated!'.
Per quanto riguarda l'entità Tag, l'inserimento avviene in modo distinto. Infatti ogni tag mappato viene inserito controllando a priori la presenza di un eventuale genitore.
Nel caso la relazione ricorsiva sia stata configurata viene in primo luogo creata l'istanza relativa al parent. Successivamente nella creazione dell'oggetto tag viene assegnato l'indirizzo del genitore come valore all'attributo *parent*.
Per questa prima versione non viene consentita la creazione di gerarchie complesse di questi oggetti. Ogni entry può essere collegata al massimo ad un'altra necessariamente senza alcun riferimento ricorsivo.
Il riferimento all'entry relativa viene creato assegnando come argomento della funzione di inserimento il puntatore all'oggetto creato.
Gli attachment invece vengono gestiti diversamente, prima di tutto ne viene controllata la consistenza dell'attributo *fullpath*, successivamente viene verificata la presenza, nel filesystem, del file relativo e, infine, caricato o meno a seconda dell'esito delle verifiche precedenti.
Questo consente di risparmiare eventuali caricamenti inutili. Infatti, come citato nella descrizione del modello dei dati, l'entità viene gestita in modo diverso da tag. Per ogni istanza la relazione 'molti a molti' con ogni altra entità viene gestita con la specifica dell'entità stessa e dell'id relativo all'entry associata.
Se venisse quindi controllata la presenza o meno dell'entry attraverso la verifica di tutti i suoi attributi lo stesso contenuto associato ad una diversa entità verrebbe interpretato come un diverso allegato e quindi caricato nuovamente aumentando la latenza di risposta.
Per l'importing viene utilizzato il campo *fullpath* specificato a livello di modello di tipo *FileField*.
Il framework django permette infatti di eseguire l'upload sul server attraverso l'uri o il percorso inserito in questo campo.
Di seguito viene riportata la funzione di importazione sviluppata:

```
def importAttachment(obj,content,name):
  obj.fullpath.save(name, File(open(content[0])), save=False)
  return
```

Il primo argomento della funzione è relativo all'istanza di Attachment, il secondo all'indirizzo della risorsa mentre il terzo e ultimo al nome con cui viene salvata nel filesystem del server.
La destinazione dell'allegato viene determinata al momento dell'inserimento dell'entry nel database. Infatti è stato specificato nella classe Attachment il codice seguente:

```
def get_path(self,filename):
      path=os.path.join('attachment',self.entity)
```

```
    return os.path.join(path,filename)

fullpath=models.FileField(null=True, upload_to=get_path)
```

La funzione get_path restituisce il percorso </attachment / *entity* / *filename*>. La variabile *path* corrisponde alla concatenazione tra la stringa 'attachment' e l'attributo *entity*. Nello stesso modo viene successivamente eseguita l'unione tra la stringa ricavata precedentemente e il nome del file dell'allegato.
In questo modo tutti gli allegati vengono caricati ordinatamente, nel filesystem del server, nelle cartelle corrispondenti alle entità associate.
A questo punto è stato illustrato il procedimento per il caricamento di tutti i dati contenuti nel foglio elettronico. Tuttavia non è stato definito l'ordine con il quale vengono analizzate e gestite le singole entità. Per l'assegnazione dei riferimenti prima di inserire tag e allegati è fondamentale venga memorizzata l'entry relativa all'entità associata, in modo tale da ricavarne l'indirizzo di memoria. Nel caso non venga attivata l'entità specifica, infatti, non possono essere attivate neppure le entità di supporto.
Inoltre le entità Item_details e Outfit sono strettamente associate ad Item. Se non viene mappato nessun attributo in questa entità non vengono quindi create. Per questo motivo il primo oggetto che viene istanziato, *item_obj*, è quello relativo all'entità principale.
Successivamente, per quanto riguarda Item_details, all'attributo *item* viene assegnato l'indirizzo di item_obj. Per Outfit invece, essendo collegata da una relazione 'molti a molti', item_obj viene aggiunto alla lista degli oggetti referenziati dall'entry specifica.
Una volta terminata l'elaborazione le tabelle create vengono memorizzate in un file csv spedito via HttpResponse al client.
Di seguito viene riportato uno screenshot relativo a una parte dell'output prodotto in uno dei test effettuati:

ITEM

ID	CODE	DESCRIPTION
1	GPelle 1	giaccone di pelle 1
2	GPelle 2	giaccone di pelle 2 D
3	Gpelle 4	giaccone di pelle
4	GPelle 2	giaccone di pelle 2 HD
5	PJeans 1	pantalone jeans 1
6	PJeans 2	pantalone jeans 2
7	PJeans 3	pantalone jeans 3
8	WPelle 1	portafoglio pelle 1
9	WPelle 2	portafoglio pelle 2
10	SLana 1	sciarpa lana 1
11	MLana 1	maglione lana 1
12	GLana 1	golf lana 1
13	CLana 1	cappello lana 1
14	PJeans 4	pantalone jeans 4
15	PJeans 4	pantalone jeans 5

TAG

ID	TYPE	NAME	PARENT ID
1	colore	marrone	
2	acrilico	80	1
3	colore	nero, bianco	
4	acrilico	27	3
5	colore	grigio	
6	acrilico	10	5
7	colore	nero	
8	acrilico	10	7
9	acrilico	30	7
10	acrilico	30	1
11	acrilico	40	7
12	colore	grigio, beige	
13	colore	blu	

Figura 26 Esempio di output dell'applicazione generatrice relativo ad Item e Tag

3.3 Conclusioni

In questo step è stata descritta l'applicazione generatrice sviluppata. Sono stati inoltre descritti nel dettaglio i metodi di elaborazione dei dati e i risultati ottenuti.
L'applicazione è stata testata con fogli di dati di dimensioni elevate, ovvero decine di migliaia di righe producendo al variare della configurazione attuata i risultati di seguito riportati.
I test sono stati compiuti su un foglio elettronico composto da 15000 righe mappandone i campi in modi distinti:

attributi per ITEM	TAG senza Parent	TAG con Parent	tempo(sec)
2	0	0	60,34
2	2	0	167,05
4	0	0	60,69
4	1	0	113,45
4	5	0	335,82
4	3	2	437,28
4	10	0	626,81
4	5	5	893,13
4	15	0	938,47
4	10	5	1227,95
4	12	8	1452,19
4	20	0	1040,27
4	14	9	2216,19
4	23	0	1648,53

È possibile notare come l'aumentare degli attributi da mappare aumenti sensibilmente il tempo di elaborazione. In particolare nelle prime due prove effettuate sono stati inseriti gli stessi attributi. Nella prima ogni set di attributi distinti viene memorizzato come una unica entry di Item nel database. Nella seconda, invece, metà dei campi vengono mappati come attributi statici di Item, mentre l'altra metà come tag. Questo comporta ad un tempo di elaborazione pari al triplo del precedente. Infatti la memorizzazione nel database compiuta dall'applicazione generatrice comporta, in questo caso specifico, un numero di accessi superiore del 50 % rispetto al precedente.
Il maggiore numero di accessi in memoria di massa comporta all'aumento di tempo verificato, essendo le operazioni di lettura/scrittura il fattore determinante del tempo di elaborazione.
Da notare inoltre come la mappatura di un numero di campi doppio implichi una lieve diminuzione delle prestazioni nel caso in cui non vengano utilizzati i tag. Questa osservazione contribuisce a consolidare l'ipotesi espressa precedentemente.
Nonostante questo i risultati ottenuti soddisfano i requisiti iniziali. Il tempo impiegato infatti è lineare con il numero dei campi mappati in tag o, in generale, con il numero di entry inserite.
Per quanto riguarda l'aumento dovuto alla memorizzazione dinamica degli attributi è da valutare il vantaggio fornito dalla configurabilità e adattabilità globali fornite con la diminuzione delle prestazione ottenute, comunque accettabili per quanto spiegato precedentemente.

L'obiettivo di questo step è stato raggiunto, l'applicazione è solida e fornisce delle prestazioni accettabili.
Il prossimo passo è ora quello di passare allo sviluppo della stessa applicazione in ambito distribuito, ovvero modificando nella progettazione l'utilizzo di un server centralizzato, come Apache, per l'elaborazione di grosse quantità di dati.

Capitolo 4

Web app generatrice su architettura Django/Google App Engine

4.1 Background

4.1.1 Cloud computing

La tecnologia *cloud computing* riguarda lo sviluppo e l'integrazione di infrastrutture informatiche basati sul protocollo TCP/IP[12]. Tali infrastrutture possono essere, per esempio, micro processori, memorie di massa, reti ad alta velocità, sistemi informativi.
Senza gli standard attuali di connessione e condivisione di dati in rete, non sarebbe possibile parlare di cloud computing, fatto che la rende un argomento di ricerca attuale.
L'idea è quella che l'utente possa accedere ai servizi indipendentemente dalla locazione fisica delle risorse o dalle caratteristiche di queste. Tali servizi inoltre vengono sviluppati e implementati su un sistema eterogeneo e distribuito.
Nel 2007 IBM e Google annunciarono la collaborazione in un progetto di ricerca nel settore del cloud computing. Da allora molte software house decisero di intraprendere la ricerca in questo settore.
I servizi di cloud computing si suddividono in tre categorie: *Infrastructure-as-a-Service*(IaaS), *Platform-as-a-Service*(PaaS) e *Software-as-a-Service*(Saas).
Infrastructure-as-a-Service è la possibilità di utilizzo da remoto in quantità elevata di risorse quali l'elaborazione dei dati, lo storage e la rete.
Considerando lo storage, per esempio, quando un utente da remoto accede ad una quantità di memoria di massa presente nel *cloud*(l'insieme distribuito dei nodi eterogenei) il servizio effettuato riguarda solo la porzione dedicata. Non è quindi necessario conoscere la locazione del disco fisso, o dei dati che vengono memorizzati, per l'utilizzo degli stessi.
Platform-as-a-Service fornisce il supporto per un insieme di applicazioni attraverso un'interfaccia utente dedicata. Rappresenta quindi la connessione tra l'hardware e l'applicazione. Questo servizio è attualmente molto richiesto e la piattaforma predominante è data da Azure Service's Platform di Microsoft. In questo contesto inoltre è stata sviluppata la piattaforma Google App Engine, che verrà presentata nella sezione successiva.
L'idea di Software-as-a-Service è la virtualizzazione delle applicazioni dell'utente. Viene tolta la necessità di installare e lanciare i programmi nel proprio personal computer. Il servizio di cloud computing eseguirà queste operazioni restituendo lo stesso risultato ad un costo minore.
Il problema principale di questi servizi risiede nell'importazione dei dati. L'uploading delle informazioni sul server distribuito, infatti, come verrà descritto nel capitolo quarto, rappresenterà il fattore predominante di inefficienza dell'applicazione sviluppata.

4.1.2 Google App Engine

Google App Engine(GAE) è un servizio di cloud computing PaaS che permette di eseguire le proprie applicazioni caricandole nell'infrastruttura Google.
Le caratteristiche principale di GAE[13] sono:

- Supporto per tutti gli standard web, anche obsoleti.
- Datastore con controllo della persistenza e possibilità di effettuare query, ordinamenti e transazioni.
- Scalabilità all'aumentare degli utenti e del volume di dati introdotto dalle richieste.

- API per l'autenticazione nel proprio account Google.
- SDK completo di simulatore per l'esecuzione locale delle applicazioni.
- Supporto multitasking.
- Scheduling dei task per attivare azioni ed eventi in istanti specifici e intervalli regolari.

Google App Engine supporta applicazioni scritte in diversi linguaggi di programmazione tra i quali Java, Python e altri ,che utilizzino per l'esecuzione dei programmi finali un interprete basato sulla Java Virtual Machine, quali per esempio JavaScript e Ruby.
Gli ambienti di sviluppo più utilizzati, quali Java e Python, sono stati forniti per assicurare un'esecuzione efficiente e che non interferisca con le altre applicazioni di sistema.
I servizi offerti sono gratuiti se utilizzati sotto una determinata soglia giornaliera.
Un'applicazione può utilizzare al massimo 500 MB di spazio su disco per la memorizzazione dei dati e banda e CPU sufficienti per servire 5 milioni di pagine al mese, in modo totalmente gratuito. Oltre questa soglia il servizio di pagamento deve essere abilitato per poter continuare ad utilizzare la propria applicazione da remoto.

4.1.3 BigTable

I database classici vengono implementati partendo da uno schema entità/relazione. Ogni entità e associazione 'molti a molti' corrisponde ad una tabella, mentre le relazioni vengono mappate con l'utilizzo di chiavi esterne e id univoci corrispondenti. Questo porta alla descrizione di una struttura leggibile e facilmente gestibile per quanto riguarda la modifica e l'inserimento dei dati, ma ,come citato nel secondo capitolo, ad un ritardo nelle risposte delle query dovuto al numero di join effettuati tra le tabelle, operazione molto costosa.
Per questo motivo il problema dei database di questo tipo, ovvero dei database relazionali, è dovuto alla scalabilità. In grosse basi di dati, per esempio quelle relative ai portali web di e-commerce o ai social network, composte da miliardi di dati, le prestazioni offerte in termini di tempo di risposta non sono infatti soddisfacenti. Inoltre la necessità di memorizzazione in cloud composti da migliaia di nodi ne rende molto complicata la gestione.
A tale scopo sono state progettate delle soluzioni alternative: i database non relazionali[15].
Essi, a differenza di quelli illustrati precedentemente, si distinguono in quanto i dati non sono memorizzati in tabelle distinte collegate secondo le associazioni definite. Idealmente ogni entry contiene tutti gli attributi associati. Questa caratteristica porta alla necessità di definire una base di dati totalmente denormalizzata con i vantaggi legati alla ricerca e all'estrazione di informazioni legati al difficile controllo dell'integrità e della consistenza.
Tuttavia è possibile specificare relazioni tra le entità definite, anche se la loro gestione avviene ad alto livello. Non vengono effettuate infatti operazioni di join da parte del dbms.
Da citare infine che la struttura delle basi di dati gestite attraverso questi dbms può essere specificata in tempo reale nel caricamento dell'applicazione, questo ne rende l'utilizzo molto flessibile.

BigTable[14] è un sistema di storage non relazionale distribuito progettato per la memorizzazione di grandi quantità di dati. La sua caratteristica principale è la scalabilità offerta. Consente infatti il salvataggio di petabyte di dati raccolti in centinaia di server. Molte applicazioni create da Google utilizzano questo sistema come dbms, incluso il motore di ricerca, Google Earth e Google Finance. Applicazioni che, in quanto ad utilizzo del dbms hanno un ruolo molto diverso, sia in termini di contenuto(URL, pagine web, immagini satellitari) e sia di latenza massima(risposte real-time o asincrone).
Per quanto riguarda la struttura il modello di dati BigTable è una mappa multidimensionale distribuita e ordinata.
L'indice della mappa è dato da una chiave di riga, una chiave di colonna e un timestamp. Ogni valore contenuto in ogni cella è un array di byte non interpretato.
Per queste caratteristiche BigTable fa parte della classe di database non relazionali *key-value*, ovvero ogni elemento è univocamente definito da una chiave e le associazioni vengono specificate come

chiavi contenute negli attributi. Inoltre la memorizzazione di ogni entità avviene nella stessa struttura definita.
Questa mappa, una volta raggiunte dimensioni prefissate, viene distribuita nel cloud suddividendone le righe, come verrà illustrato successivamente.

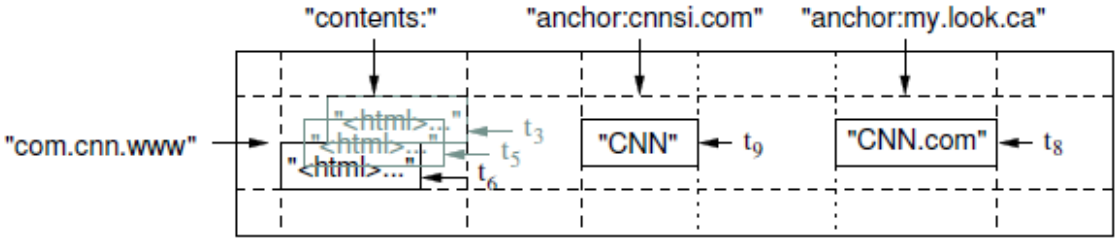

Figura 27 esempio di riga BigTable

Le chiavi di riga sono stringhe arbitrarie. Ogni operazione di lettura o scrittura compiuta utilizzando una chiave di riga è atomica, qualsiasi sia il numero di attributi o colonne presenti.
BigTable mantiene i dati ordinati in ordine lessicografico per chiave di riga.
Per la distribuzione e la partizione tra i server viene stabilito, come citato in precedenza, un numero di righe che costituisca l'unità di caricamento della base di dati, il *tablet*. Mentre il modello logico viene mappato nel database fisico e replicato in ogni data center del cloud in modo tale possano essere riconosciute e gestite le righe del tablet.
Le chiavi di colonna vengono raggruppate in insiemi chiamati *famiglie*. Esse formano l'unità di base per il controllo d'accesso.
Le famiglie contengono dati omogenei e devono essere dichiarate prima che qualsiasi dato venga memorizzato utilizzando una qualsiasi chiave di colonna appartenente.
L'idea è quella di mantenere basso il numero di famiglie presenti consentendo comunque un numero illimitato di colonne.
Ogni cella in BigTable può contenere versioni multiple dello stesso dato. Esse vengono indicizzate da un timestamp rappresentato da un intero a 64 bit.
Il riferimento temporale può essere assegnato automaticamente dal dbms al momento dell'inserimento del dato nella cella o assegnato dall'utente. In questo modo vengono evitate le collisioni. Versioni differenti vengono salvate, infatti, con timestamp decrescenti in modo tale il dato più recente sia il primo ottenuto. Inoltre attraverso l'indicazione dell'istante di inserimento è possibile ricavare informazioni sui dati obsoleti presenti. BigTable a tal proposito consente all'utente di definire le versioni da mantenere e quelle da passare al garbage collector.
Lo creazione dei cluster BigTable è iniziata nel 2005, l'anno successivo sessanta progetti utilizzavano questo dbms. In particolare le prestazioni e l'affidabilità fornite da questo servizio, unite alla capacità di aumentare la quantità di dati inseriti, semplicemente aggiungendo nuove risorse hardware al sistema, hanno portato questa infrastruttura al centro di un progetto di ricerca e sviluppo attualmente ancora in corso.

4.2 Implementazione

4.2.1 Strumenti utilizzati

Lo sviluppo di questa parte è stato compiuta sulla stessa macchina presentata nel progetto precedente. In essa è stato installato l'SDK fornito da Google: *Google App Engine Launcher*.
Il software development kit contiene, oltre all'interprete python e le librerie fornite, un'interfaccia utente che permette di eseguire in locale la propria applicazione.
Fornisce inoltre una interfaccia ottimizzata per:

- Visualizzare, inserire o eliminare gli oggetti presenti nel datastore.
- La possibilità di scrivere codice in una console interattiva.
- Visualizzare e eliminare oggetti presenti nella memoria cache.
- Avviare o fermare l'esecuzione di un elemento dalla lista dei task associati ad ogni coda.
- Avviare o fermare l'esecuzione di un elemento dalla lista dei task che vengono eseguiti ad istanti o intervalli pre fissati.
- Visualizzare o eliminare un elemento dall'insieme dei messaggi ricevuti o delle comunicazioni real-time in generale effettuate utilizzando il protocollo XMPP(Extensible Messaging).
- Visualizzare o eliminare un elemento dalla lista delle mail ricevute o inviate dall'applicazione.
- Effettuare il porting dell'applicazione nel cloud Google.

Per l'IDE di sviluppo è stata mantenuta la scelta di Eclipse Helios, in quanto fornisce totale supporto per l'utilizzo dell'SDK. Il linguaggio di programmazione utilizzato è stato python.
Nonostante GAE fornisca il totale supporto per il framework django la versione standard fornita è la 0.96.
Essendo la versione utilizzata per lo sviluppo dell'applicazione generatrice la 1.2.3 è stato optato di includere quest'ultima nell'applicazione eseguita nel cloud utilizzando lo stumento *Google App Engine Helper For Django.* Il procedimento nei dettagli verrà illustrato nella sezione relativa alla creazione dell'applicazione base nella piattaforma di sviluppo.

4.2.2 Requisiti e obiettivi

Nel capitolo precedente è stata sviluppata l'applicazione generatrice utilizzando l'architettura Django/Apache, utilizzando quindi un'elaborazione centralizzata che ha portato a dei risultati molto soddisfacenti come illustrato.
Il prossimo step riguarda lo sviluppo della stessa applicazione su un sistema di calcolo distribuito, Google App Engine. Il sistema, rilasciato nella versione 1.4.1, permette di caricare nei server l'applicazione creata e, una volta inserita nel cloud, di eseguirla ed effettuarne il testing delle prestazioni.
Proprio per il fatto che si tratta di una piattaforma di recente sviluppo non si trovano ancora delle valutazioni tecniche al di fuori di quelle fornite dalla stessa software house produttrice.

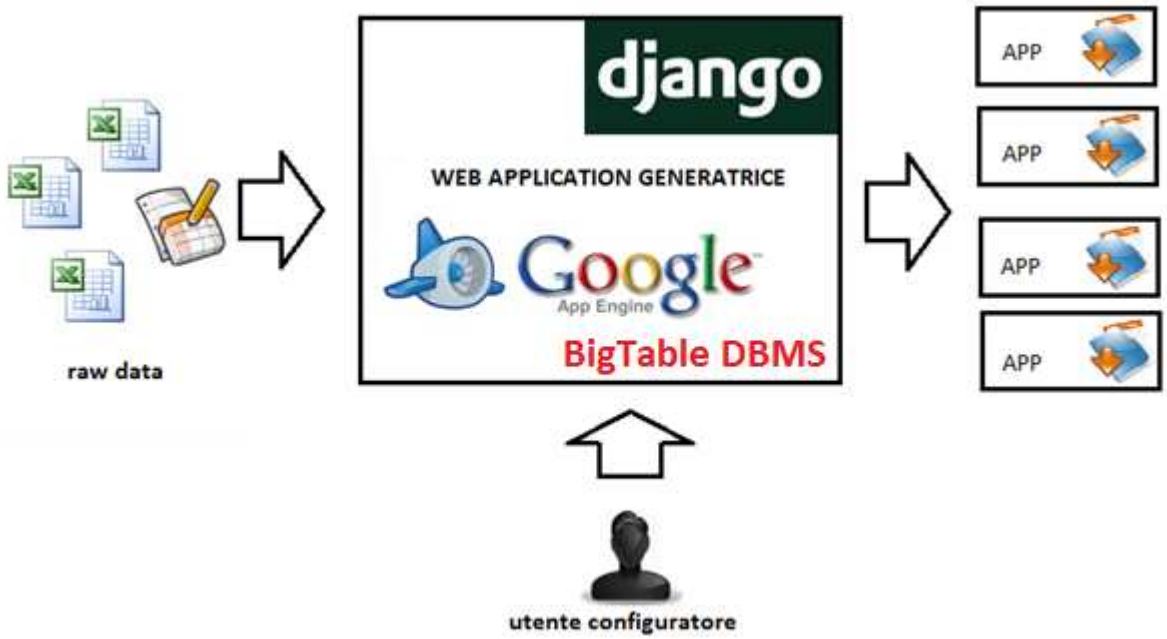

Figura 28 schema applicazione generatrice architettura Django/Google App Engine

È stato quindi programmato uno studio approfondito delle possibilità e delle limitazioni offerte da questo servizio nell'ambito dello sviluppo della web application generatrice. Studio che comprende l'analisi e l'implementazione dell'applicazione in questa nuova struttura e che verrà descritto in questo capitolo.

4.2.3 Differenze tra sviluppo in Apache/SQLite e Google App Engine/BigTable

Le differenze sostanziali risiedono nel dbms e nel sistema di gestione dei file utilizzati. Infatti, come illustrato nella sezione iniziale, GAE utilizza un dbms non relazionale. Questo comporta alla ridefinizione totale per quanto riguarda la parte centrale dell'applicazione generatrice, ovvero l'inserimento dei dati.
Per l'importazione, invece, App Engine non permette, come è stato invece realizzato nel progetto illustrato nel capitolo precedente, di salvare nel filesystem del server spreadsheet e attachment. In particolare viene sollevata un'eccezione anche nel caso venga tentato di aprire un file con i permessi di scrittura.
Il motivo è dato dagli obiettivi di sicurezza posti da Google per l'esecuzione delle applicazioni distribuite.
Tali applicazioni vengono eseguite in un ambiente che fornisce accesso limitato al sistema operativo sottostante. Queste limitazioni permettono a GAE di distribuire le richieste web su nodi eterogenei. L'applicazione viene quindi isolata in un ambiente sicuro e affidabile indipendente dal sistema operativo e dall'hardware.
Esempi di queste limitazioni sono:

- L'accesso ad altre macchine nella rete può avvenire solo attraverso servizio e-mail o specifica dell'url. Non è possibile come in Apache specificare l'indirizzo locale del file da includere. La connessione con queste può essere realizzata solamente effettuando richieste http(o https) sulle porte standard.
- Un'applicazione non può creare nuovi oggetti nel filesystem. È possibile la lettura di file solo se caricati nel server con l'applicazione stessa. Per altri caricamenti devono essere utilizzati i servizi forniti quali il datastore, memcache, ecc..

Da citare inoltre le quote imposte nell'elaborazione dati. Il tempo massimo di esecuzione di ogni singolo task è di trenta secondi. Se il tempo di processing eccede tale limite il task viene fatto ripartire da capo con tutte le problematiche legate all'idempotenza e alla latenza di risposta.
Rispetto alla versione centralizzata deve essere quindi considerato un meccanismo di elaborazione parallela che consenta la suddivisione in task.

Riassumendo i problemi principali dovuti alla struttura distribuita riguardano:

- La gestione della base di dati
- L'importazione dei contenuti
- L'elaborazione dei dati importati

Questi tre aspetti costituiscono il corpo dell'applicazione. Lo studio e lo sviluppo di implementazioni adatte alla struttura di GAE rappresentano quindi una totale rielaborazione di quanto svolto e illustrato nel capitolo precedente. Le sezioni successive a questo proposito forniranno una descrizione dettagliata dei cambiamenti attuati per il porting nella struttura distribuita.

4.2.4 Configurazione e inizializzazione nella struttura distribuita

La prima operazione compiuta per lo sviluppo dell'applicazione distribuita consiste nella creazione di una web application django-based non contenente alcuna implementazione. GAE infatti supporta,

come già citato, una versione di django obsoleta il cui aggiornamento è necessario ai fini degli obiettivi finali.
Il problema a prima vista sembra risolvibile semplicemente includendo la libreria relativa nell'applicazione caricata. Il framework però nella sua versione standard supporta solo database relazionali e non permette quindi, come illustrato nella parte di background, di utilizzare BigTable.
La soluzione consiste nell'utilizzo di una ulteriore applicazione django che utilizzi il framework di default per configurare le librerie di sistema dichiarando la versione 1.2.3.
Per illustrare nel dettaglio questo procedimento è necessario fornire la struttura della web application base distribuita.
La prima operazione consiste nella creazione del progetto contenente il file python relativo al main thread e il file di configurazione: *app.yaml*. In quest'ultimo devono essere specificati i dati identificativi dello script caricato.
Di seguito viene riportata la struttura di un file yaml generico:

```
application: application_name
version: 1.0
runtime: python
api_version: 1
handlers:
- url: /.*
  script: main_thread.py
```

Dove *application_name* è l'identificativo univoco per l'applicazione una volta caricata nel cloud. Rappresenta quindi l'id di registrazione nel servizio App Engine. *Version* invece è l'identificativo per la versione dell'applicazione. È possibile infatti caricare nel server differenti versioni che vengono distinte e richiamate grazie a questo parametro.
Successivamente vengono elencati i campi relativi all'esecuzione dello script indicato nell'ultima riga.
Con queste informazioni è possibile lanciare l'applicazione in locale utilizzando l'SDK fornito o caricare l'applicazione in remoto effettuando quindi il *deploy*.
In ogni caso verrà eseguito lo script main_thread e visualizzato in output il risultato. È stata quindi creata una prima applicazione GAE.
Il prossimo passo consiste nella configurazione django-based all'ultima versione rilasciata. A questo scopo viene utilizzato il pacchetto già citato: Appengine Helper for Django.
Esso contiene al proprio interno:

- I file relativi all'impostazione base di un progetto django e illustrati nel capitolo precedente.
- La cartella *appengine_django*.
- I file app.yaml e main.py.

La struttura è relativa ad un generico progetto django ma anche ad una applicazione portabile su App Engine.
Aggiungendo nel progetto la libreria aggiornata del framework e l'applicazione django-based è possibile eseguire nella piattaforma distribuita quest'ultima senza alcun problema di conflitto tra le versioni presenti. Per fare questo il file settings.py che, come descritto nel capitolo precedente, contiene i dati di inizializzazione, deve dichiarare le web application appengine_django e la django-based.
Deve essere impostato inoltre il riferimento al dbms BigTable. L'applicazione di supporto provvede a caricare le librerie aggiornate consentendo quindi alla django-based di utilizzarle nella propria esecuzione.
A questo punto è possibile cominciare l'implementazione dell'applicazione generatrice partendo dal modello dei dati.

4.2.5 Base di dati globale

Nella sezione dedicata alle differenze tra le architetture di elaborazione utilizzate è stata sottolineata la mancanza di un database relazionale e quindi del supporto ORM fornito da django.
È necessario quindi definire un nuovo modello logico partendo da quello sviluppato precedentemente. Per la costruzione del database App Engine supporta lo sviluppo del modello logico fornendo api specifiche. La costruzione risulta quindi molto simile a quella operata e supportata da django. Ovviamente in questo caso la configurazione è relativa al datastore BigTable.
Per definire un modello di dati deve essere creata una classe associata specificandone gli attributi e assegnando ad ognuno di questi un'istanza della classe relativa al tipo di dato che deve essere memorizzato e gestito. La classe relativa all'importazione dei modelli è *google.appengine.ext.db*.
Gli oggetti utilizzati per definire i valori dei parametri sono:

- db.IntegerProperty per i valori numerici interi.
- db.FloatProperty per i valori decimali.
- db.StringProperty per le stringhe.
- db.BooleanProperty per i valori booleani.
- db.DateTimeField per la memorizzazione di istanti temporali.

Per quanto riguarda le relazioni tra i modelli implementati è possibile associare ad un dato attributo una *chiave* specifica. Ad ogni oggetto istanza di un modello dichiarato, una volta memorizzato nel datastore, corrisponde una chiave identificativa univoca. Essa può essere utilizzata per ricavare un oggetto dato il valore dei suoi attributi dal database.
L'associazione da un'istanza ad un'altra nel modello logico viene quindi realizzata con l'assegnamento dell'oggetto chiave. Per questo motivo le relazioni supportate da App Engine sono:

- 'uno a molti': Per questa relazione l'istanza del modello lato 'molti' deve contenere la chiave relativa all'istanza del modello lato 'uno' come attributo di tipo db.ReferenceProperty().
- ricorsiva: Viene specificato come valore dell'attributo chiave il riferimento ad un altro oggetto dello stesso modello attraverso la creazione dell'oggetto istanza di db.SelfReferenceProperty.

Non viene invece supportata la relazione 'molti a molti'. Questo porterà a delle modifiche fondamentali per l'associazione tra i modelli Item e Outfit. Per la realizzazione di questa relazione è stata creato un modello intermedio, Item_Outfit, collegato da una relazione 'molti a uno' sia con Item che con Outfit.

Viene di seguito riportata la traduzione del modello logico relativo al database Global per l'infrastruttura distribuita:

```
class Base(db.Model):
   deleted=db.BooleanProperty()
   language=db.StringProperty()
   position=db.IntegerProperty()
   valid_from=db.DateTimeProperty()
   valid_to=db.DateTimeProperty()
   last_update=db.DateTimeProperty()
   note=db.StringProperty()
   def __unicode__(self):
      return self.id
   class Meta:
      abstract=True

class Tag(db.Model):
   type=db.StringProperty()
```

```
    name=db.StringProperty()
    parent_tag=db.SelfReferenceProperty()
    user=db.StringProperty()
    def __unicode__(self):
        return self.id

class Attachment(db.Model):
    content_type=db.StringProperty()
    attachment_entity=db.StringProperty()
    attachment_entity_id=db.IntegerProperty()
    attachment_attribute=db.StringProperty()
    type=db.StringProperty()
    filename=db.StringProperty()
    fullpath=db.StringProperty()
    file=db.BlobProperty()
    user=db.StringProperty()
    def __unicode__(self):
        return self.id

class Item(Base):
    code=db.StringProperty()
    description=db.StringProperty()
    model_code=db.StringProperty()
    model_description=db.StringProperty()
    tags=db.StringListProperty()
    user=db.StringProperty()
    def __unicode__(self):
        return self.code

class Outfit(Base):
    code=db.StringProperty()
    description=db.StringProperty()
    tags=db.StringListProperty()
    user=db.StringProperty()
    def __unicode__(self):
        return self.id

class Item_Outfit(db.Model):
    item=db.ReferenceProperty(Item)
    outfit=db.ReferenceProperty(Outfit)
    user=db.StringProperty()
    def __unicode__(self):
        return self.id

class Item_details(Base):
    sku=db.StringProperty()
    bar_code=db.StringProperty()
    size_code=db.StringProperty()
    size_description=db.StringProperty()
    price=db.FloatProperty()
    item=db.ReferenceProperty(Item)
    tags=db.StringListProperty()
    user=db.StringProperty()
    def __unicode__(self):
        return self.id

class Interface_image(Base):
    image=db.StringProperty()
```

```
    tags=db.StringListProperty()
    def __unicode__(self):
        return self.id

class Translation(Base):
    translation=db.StringProperty()
    translation_code=db.StringProperty()
    translation_language=db.StringProperty()
    translation_key=db.StringProperty()
    tags=db.StringListProperty()
    def __unicode__(self):
        return self.id
```

Per quanto riguarda la gestione della base di dati, App Engine, come già citato, non supporta l'ORM completo e affidabile illustrato nella sezione precedente. È quindi necessario scrivere manualmente le query di accesso al database nel linguaggio GQL.
GQL è un linguaggio SQL-like per recuperare entry o chiavi dal datastore di GAE le cui caratteristiche proprie sono differenti da quelle di un linguaggio di specifica di query per un database relazionale. È sufficiente pensare, infatti, all'assoluta mancanza di join. Nonostante questo la sintassi con la quale viene fornito è molto simile a quella di SQL.
Ogni GQL query ritorna zero o più entry o oggetti di tipo Key dell'entità richiesta. Viene di seguito fornita una query utilizzata per la ricerca di un oggetto Item avente per attributi i dati ricavati da una generica riga dello spreadsheet:

```
 query=db.GqlQuery("SELECT __key__ FROM Item WHERE code=:1 AND description=:2 AND
model_code=:3 AND model_description=:4 AND user=:5",
x[item['code']],x[item['description']],x[item['model_code']], x[item['model_description']], dictionary['user'])
if query.fetch(1)==[]:
        item_obj.put()
        self.item_key=item_obj.key()
else:
        self.item_key=query.fetch(1)[0]
```

La query recupera le chiavi relative alle entry della tabella Item composte dagli attributi elencati. Se il risultato è una lista vuota allora l'elemento non è presente e viene quindi inserito. Altrimenti viene caricata la chiave del primo(e unico) elemento trovato per essere utilizzata nell'eventuale memorizzazione di altre entità collegate ad Item.
Per quanto riguarda l'inserimento di Outfit, in mancanza della possibilità di effettuare operazioni utilizzando l'operatore di join viene compiuta una ricerca manuale sulle tabelle Item_Outfit e Outfit:

```
query_outfit=db.GqlQuery("SELECT __key__ FROM Outfit WHERE code=:1 AND description=:2 AND
user=:3", x[outfit['code']], x[outfit['description']], dictionary['user'])

if query_outfit.fetch(1)==[]:
        outfit_obj.put()
        self.outfit_key=outfit_obj.key()
else:
        self.outfit_key=query_outfit.fetch(1)[0]

item_outfit_obj=Item_Outfit(item=self.item_key, outfit=self.outfit_key, user=dictionary['user'])
query_item_outfit=db.GqlQuery("SELECT __key__ FROM Item_Outfit WHERE item=:1 AND outfit=:2 AND
user=:3", self.item_key, self.outfit_key, dictionary['user'])

if query_item_outfit.fetch(1)==[]:
        item_outfit_obj.put()
```

La prima parte controlla se l'oggetto Outfit è già presente nel datastore eventualmente inserendolo e recuperando la relativa chiave. Viene successivamente controllata la presenza della coppia di chiavi di Item e Outfit in Item_Outfit. Nel caso non sia presente anche in questo caso avviene l'inserimento nel database.
Da notare che l'operazione in termini di risorse CPU è molto costosa. Per questo motivo le relazioni 'molti a molti' non sono direttamente supportate da GAE. Una versione ottimizzata dell'applicazione generatrice deve quindi prevedere un'ulteriore denormalizzazione delle tabelle del datastore.

Tornando al modello logico, come è stato illustrato rispetto alla versione centralizzata sono stati modificati diversi aspetti. In primo luogo è stato aggiunta la classe Tag in quanto la libreria utilizzata era stata implementata per un sistema totalmente relazionale ed è stato quindi deciso di procedere in un altro modo. Inoltre nella classe Attachment l'attributo *fullpath* viene gestito come una semplice stringa e non come un *FilePathField* in quanto non supportato come tipo dal dbms. Da citare infine che il modello logico finale comprende altre classi che sono state implementate per risolvere determinati problemi che verranno discussi successivamente.
Nelle prossime sezioni verranno presentate le estensioni compiute sul modello logico e le motivazioni che hanno portato ad inserire le classi aggiuntive.

4.2.6 Gestione dei tag

Per quanto riguarda la gestione dei tag nel progetto sviluppato in Apache era stata riscontrata una latenza molto maggiore per quanto riguarda l'inserimento di istanze di questo tipo. In particolare memorizzando il contenuto in una unica entry, piuttosto che aggiungere gli attributi dinamici corrispondenti, il tempo di risposta diminuiva sensibilmente.
L'inserimento coinvolgeva infatti tre diversi modelli: Tag, TaggedItem e il modello dell'istanza di riferimento. Essendo questa operazione risultata molto costosa è stato deciso di realizzare il procedimento di *tagging* utilizzando solamente il modello Tag e l'istanza di riferimento.
In altri dbms questo non sarebbe possibile a meno di introdurre una grossa quantità di ridondanza. Per ogni tag collegato ad una determinata entry dovrebbero essere infatti inserite delle copie di quella stessa istanza contenenti l'attributo tag assegnato.
La caratteristica di non relazionalità implica quindi una latenza ancora maggiore per la memorizzazione e la gestione congiunta di un modello logico di questo tipo.
Tuttavia BigTable offre una soluzione innovativa: attributi di tipo *db.StringListProperty*. Questa proprietà consente la memorizzazione di una lista di stringhe ASCII o unicode.
I valori contenuti nelle liste del tipo citato sono indicizzati e possono essere utilizzati quindi per operazioni di filtraggio e di ordinamento.
L'idea è quella di creare una istanza del modello tag ad ogni necessità associandone la chiave o l'id come elemento della lista di stringhe associata all'istanza del modello generico taggato.
Per esempio associare ad un Item 'pantalone' i tag 'colore' e 'acrilico' corrisponde al seguente risultato.

TAG

Id	Type	name	parent
1	colore	grigio	None
2	acrilico	20%	1

ITEM

Id	code	description	model_code	model_description	tags
1	1	pantalone	11	pantalone jeans	[1,2]

Nel progetto è stato scelto di utilizzare per la visualizzazione del database gli id degli oggetti associati. In realtà, come è stato illustrato, i riferimenti vengono realizzati attraverso attribuzione di chiavi specifiche come attributi. L'id univoco viene comunque assegnato automaticamente dal dbms al

momento dell'inserimento di queste. Dall'oggetto di tipo *Key* associato all'istanza è possibile, infatti, derivare il valore dell'identificativo e inserirlo successivamente come attributo dell'entità di riferimento.
La gestione dei tag compiuta in questo modo presenta dei miglioramenti in quanto la ricerca è molto più veloce. In accordo quindi con gli obiettivi imposti. Tuttavia l'inserimento presenta alcune problematiche rilevanti, esse riguardano il controllo dell'integrità della base di dati, punto debole dei dbms non relazionali.
Nell'inserimento di nuovi tag per un determinato oggetto la lista relativa deve essere ovviamente aggiornata.
Il linguaggio di programmazione utilizzato, python, consente di gestire gli elementi di una lista generica in modo molto semplice.
Viene riportato di seguito il metodo di aggiornamento implementato:

```
def updateList(old_list,tag_list):
    new_list=[]
    for x in old_list:
       if x in tag_list:
          tag_list.remove(x)
       new_list.append(x)
    new_list.extend(tag_list)
    return new_list
```

Per ogni oggetto replicato nello spreadsheet, ma con valori degli attributi mappati come tag distinti inseriti in *tag_list*, viene recuperata la lista associata obsoleta *old_list*. Per evitare di inserire riferimenti duplicati la prima operazione che viene compiuta è quella di controllare la presenza, di ogni elemento di old_list, in tag_list. Se tale riscontro ha esito positivo l'oggetto viene rimosso da quest'ultima. Ogni elemento della lista obsoleta, presente o meno tra i nuovi tag, viene inserito in una nuova lista *new_list*.
Alla fine del ciclo la lista new_list viene estesa con i tag presenti in tag_list, che sono quindi quelli totalmente inediti per l'oggetto associato, e restituita in output per essere memorizzata nel modello logico e nella base di dati.

4.2.7 Importing data

GAE, come citato nelle sezioni precedenti, permette il caricamento da file attraverso specifica dell'url o inserimento in una form apposita di upload.
Il problema centrale, tuttavia, risiede nel fatto che ogni file non possa superare la dimensione massima di 1 MegaByte. Inoltre il tempo di risposta massimo di ogni richiesta è di trenta secondi. Questo per evitare di sovraccaricare i server nell'istradamento delle richieste.
Gli spreadsheet e gli allegati utilizzati nel corso delle analisi compiute nel capitolo precedente superavano abbondantemente il limite della grandezza massima. Mentre le reti congestionate possono aumentare la latenza massima prevista, rendendo quindi il caricamento a volte non realizzabile e soprattutto dal risultato aleatorio.
Per il caricamento dei file sono state analizzate diverse tecniche delle quali viene di seguito riportato un elenco dettagliato. Nell'ambito di questo progetto ne sono state utilizzate solo alcune che verranno quindi illustrate ad un livello molto più approfondito e richiamate in seguito.

- Blob: Come illustrato nella sezione relativa alla base di dati nella descrizione del modello logico gli attributi vengono mappati come oggetti *Property* relativi alla classe *db*. Una delle proprietà utilizzabili è *BlobProperty*.
  L'oggetto istanza di db.BlobPropery consente la memorizzazione e la gestione di qualsiasi dato serializzabile.

Ovvero per il salvataggio e la gestione di qualsiasi tipo di file viene assegnato all'oggetto il contenuto del file stesso. Questa soluzione è efficace ed efficiente. Inoltre viene utilizzato il datastore App Engine e quindi la velocità nel servire le richieste e la scalabilità sono caratteristiche garantite.
Il punto debole di questo procedimento è dato dalla grandezza massima dei file memorizzabili, attualmente di 1 MB. Un'idea per la soluzione consiste nel creare un array di attributi di tipo BlobProperty ed impostare un protocollo per la suddivisione reversibile del file. Purtroppo però le variabili stesse in python consentono la memorizzazione di dati di dimensione massima di 1 MB, stesso limite imposto inoltre per la dimensione degli oggetti request/response http.
Quindi la gestione di file memorizzati in variabili separate deve essere gestita a livello applicativo con tutte le complicazioni relative.

- BlobStore: Non consentendo un accesso diretto al filesystem GAE nelle ultime release consente di utilizzare un servizio wrapper per l'allocazione di file di grandi dimensioni: *BlobStore*.
  Le API messe a disposizione da questo servizio consentono all'applicazione di utilizzare oggetti *blob* caricati nel server. La caratteristica di questi consiste nella grandezza massima consentita. Al contrario degli oggetti del datastore, infatti, viene permessa la memorizzazione di dati fino a 2 GB di grandezza complessiva.
  I blob vengono creati attraverso richieste http. Tipicamente queste vengono generate per il caricamento di file esterni attraverso form composte da un campo file upload. Una volta che la form è stata inviata il servizio BlobStore crea un nuovo blob dal contenuto del file caricato restituendo la chiave associata per il riferimento nel corso dell'applicazione.
  Il rilassamento dei vincoli porta a considerare la soluzione illustrata dal punto di vista del caricamento dei dati. Tuttavia ancora una volta sussistono dei vincoli molto limitativi che non permettono di utilizzare al meglio questa tecnologia.
  In primo luogo la dimensione delle richieste resta sempre fissa al massimo ad 1 MB. Quindi per caricare un file di grandi dimensioni è ancora necessario dividerlo in segmenti inviando ciascuno di essi in un pacchetto separato.
  La realizzazione di questo script non richiede competenze particolari ed è stata realizzata con semplicità. Purtroppo però il servizio di Blobstore viene abilitato solo dopo aver stipulato un contratto di pagamento con l'azienda produttrice dell'infrastruttura distribuita.
- Bulkloader: Lo strumento Bulk Loader permette di eseguire l'upload e il download dal proprio datastore. È possible infatti caricare il proprio file csv o xml o scaricare su questo il contenuto del datastore. Permette quindi di eseguire le stesse operazioni richieste nel processing dell'applicazione generatrice e sembra a prima vista ottimo per lo scopo del progetto. Purtroppo però detiene dei limiti sensibili. In primo luogo il file è statico e quindi deve essere caricato con l'applicazione nel cloud di GAE. In termini commerciali la caratteristica descritta implica il rilascio dell'intera applicazione al cliente come prodotto open-source. Inoltre non sarebbe permesso all'utente configuratore di scegliere il file da caricare.
  Infine non permette di associare clausole al di fuori dell'inserimento nel datastore. A tale scopo sarebbe quindi necessario un processo esterno che, ad inserimento ultimato, ne controlli l'integrità. Considerando i controlli compiuti per i tag e le operazioni aggiuntive di supporto quali, per esempio, l'upload degli attachment e il controllo della compatibilità delle espressioni regolari, l'implementazione di questo processo richiede necessariamente uno studio molto approfondito.
  Non è da escludere tuttavia un utilizzo di questa tecnologia nelle future versioni per aumentare l'efficienza e le prestazioni offerte dall'applicazione.
- File statici: Un'altra idea per il caricamento consiste nella creazione di cartelle specifiche al progetto che viene distribuito in App Engine. Inserendo i file richiesti in queste cartelle sarà possibile utilizzarli, infatti, nell'esecuzione dell'applicazione distribuita. Ovviamente questo metodo non è assolutamente adatto agli scopi del progetto. Come nel caso precedente l'utente

deve poter scegliere nel corso dell'esecuzione gli spreadsheet, i file di configurazione e gli allegati da associare.

Dopo aver analizzato le soluzioni possibile e riscontrato i vantaggi e soprattutto gli svantaggi introdotti da ciascuna è stato scelto di memorizzare gli allegati nel datastore.
Per il caricamento del foglio elettronico è stato scelto, invece, di utilizzare il servizio fornito da Google Spreadsheet. Esso fa parte della classe di servizi forniti da Google Docs e consente la totale manipolazione dei dati presenti nei fogli di calcolo.
In altre parole permette di eliminare, leggere inserire o modificare singole righe o singole celle presenti nel documento.
L'idea è stata quella di prelevare un numero fissato di righe per iterazione utilizzando le API fornite e, successivamente, avviare l'elaborazione dei dati su queste.
Di conseguenza è stato tolto l'uploading da locale, ogni spreadsheet caricato in App Engine deve essere presente nel proprio account Google Documents. Il motivo di questa scelta è determinato dall'ipotesi che la latenza tra le richieste generate dal cloud Google e le risposte provenienti dal servizio Google Docs sia molto bassa.
Per quanto riguarda l'autenticazione compiuta è stato utilizzato lo stesso procedimento descritto nel corso del progetto precedente. L'unica differenza è data dalla memorizzazione del token. Non essendo fornito da GAE supporto per le variabili di sessione la memorizzazione è stata effettuata nel datastore. A questo scopo è stata aggiunta nel modello logico l'entità Token.

```
class Token(db.Model):
   token_str=db.StringProperty()
   enabled=db.BooleanProperty()
   user=db.StringProperty()
   def __unicode__(self):
      return self.id
```

Dove l'attributo *token_str* rappresenta la serializzazione del token ottenuto dall'autenticazione, mentre *enabled* indica se tale token è valido o meno. Per questioni di testing e debugging infatti tutti i token vengono registrati. Al momento di ogni nuova autenticazione da parte dello stesso utente viene quindi specificato quale scegliere tra questi nel recupero delle informazioni dall'account. I token utilizzati precedentemente infatti potrebbero essere scaduti.
Per quanto riguarda *user*, esso consiste in un attributo per il supporto della multi utenza. Infatti, nell'implementazione precedente, non veniva supportato l'utilizzo da parte di utenti distinti contemporaneamente in quanto gli obiettivi erano l'efficienza e il corretto comportamento dell'elaborazione dei dati.
Per questa parte del progetto è stato deciso di introdurre questa caratteristica aggiuntiva. Sono state delineate due idee distinte: inserire, come valore attribuito a user, l'indirizzo IP dell'utente o il nome dell'account derivato dall'autenticazione.
Per le ipotesi iniziali ogni utente dell'applicazione generatrice appartiene ad un'azienda distinta quindi è stato utilizzato l'indirizzo di rete. Questa scelta non è comunque la migliore in quanto utenti della stessa azienda, che condividono quindi uno stesso proxy, vengono associati a due indirizzi IP uguali.
Una volta caricato il token le operazioni successive vengono eseguite specificandone la stringa identificatrice nelle richieste al server.
Utilizzando successivamente le credenziali per l'accesso al server ottenute attraverso l'autenticazione viene fornita la lista degli spreadsheet presenti.
A scelta compiuta il documento non viene caricato nel server come nel caso precedente. Se superasse certe dimensioni o la latenza fosse maggiore al limite massimo consentito verrebbe infatti sollevata un'eccezione.
A questo scopo è stata aggiunta un'entità nel modello logico apposita:

```
class Csv(db.Model):
```

```
filename=db.StringProperty()
enabled=db.BooleanProperty()
csv_key=db.StringProperty()
user=db.StringProperty()
def __unicode__(self):
   return self.id
```

Essa consente di salvare nel datastore tutte le informazioni necessarie per il recupero di qualsiasi parte dello spreadsheet associato attraverso l'attributo *csv_key*. Nella sezione relativa all'elaborazione dati di questo capitolo verrà successivamente illustrato come le singole righe vengano prelevate dal foglio di calcolo.

4.2.8 Configurazione

Una volta effettuta l'autenticazione e memorizzato il riferimento nel datastore l'esecuzione passa nella configurator view. Questa vista, descritta nel capitolo precedente, permette di inserire le clausole di mapping del documento inserito.
Nello step precedente veniva caricato l'intero documento in memoria centrale e, successivamente, estratta la prima riga per la creazione della form. In questo caso invece è stato scelto di caricare dal server Google Docs solo la prima riga rendendo quindi il passaggio tra le due viste immediato. Per la configurazione è sufficiente infatti l'elenco delle classi di rappresentanza dello spreadsheet, contenuto nella riga caricata.

Per conseguire gli obiettivi di configurabilità posti in ambito di analisi progettuale è stata implementata, come nel caso dell'applicazione Apache, la possibilità di serializzazione della form configurata.
La differenza tra le implementazioni eseguite consistono nella memorizzazione compiuta.
Mentre la form atta alla configurazione del riempimento del database non ha presentato alcuna problematica nel porting in App Engine grazie al supporto per l'ultima versione django, il procedimento di serializzazione descritto nel capitolo precedente non è implementabile.
In questo caso infatti per i motivi elencati nelle sezioni precedenti non è consentita l'apertura di file in scrittura.
Fortunatamente la libreria simplejson, presentata e illustrata nel capitolo precedente, consente di effettuare la serializzazione da dizionario a file json ma anche da dizionario a stringa. È sufficiente infatti utilizzare il metodo *simplejson.dumps*.
Esso restituisce la stringa relativa al contenuto del file json associato al dizionario inserito come parametro.

```
response=HttpResponse(str(simplejson.dumps(request.POST, ensure_ascii=False)),
mimetype='application/json')
response['Content-Disposition']='attachment; filename= stream_json.json'
return response
```

Viene quindi istanziato un oggetto HttpResponse contenente il dizionario serializzato.
Successivamente viene specificato che la risposta deve essere fornita come allegato con il nome *stream_json.json*.
Per quanto riguarda la caratteristica di reversibilità richiesta alla serializzazione viene utilizzata l'istruzione *loads* di simplejson.

```
dict=simplejson.loads(request.FILES['file'].read())
```

Essa permette di caricare il dizionario relativo alla configurazione dalla stringa data dal contenuto del file json. Successivamente l'inserimento dei dati presenti nel dizionario nella form avviene nello stesso modo descritto nella sezione dedicata del terzo capitolo.

4.2.9 Processing dei dati

L'elaborazione compiuta consiste nella memorizzazione dei dati nello spreadsheet selezionato dall'utente e dell'eventuale upload degli allegati relativi. Come citato nella sezione dedicata alle differenze tra App Engine e Apache riscontrate nel porting esse si concentrano su questi argomenti.
Per quanto riguarda l'importazione dei dati dallo spreadsheet è stato illustrato come, a differenza dell'implementazione precedente, non sia possibile importare file di dimensioni consistenti in un'unica richiesta. Per questo motivo il caricamento dei dati dallo spreadsheet avviene per *segmenti*, ovvero una volta compiuta l'autenticazione vengono recuperati degli insiemi di righe contigue che vengono elaborati in modo indipendente evitando quindi l'eccezione dovuta alla latenza di risposta nell'importazione dei dati.
Per il caricamento delle singole righe dallo spreadsheet remoto deve essere utilizzato il servizio Google Spreadsheet. Di seguito vengono riportate le istruzioni per il recupero del token di sessione e la creazione di uno spreadsheet service per il recupero delle informazioni e dei dati dal file:

```
gd_client = gdata.spreadsheet.service.SpreadsheetsService(source='hUmus-data_app-1.0')
query=db.GqlQuery("SELECT __key__ FROM Token WHERE enabled=True AND user=:1",
dictionary['user']) token=Token.get(query.fetch(1)[0])
gd_client.SetAuthSubToken(token.token_str)
```

Come citato in precedenza Google mette a disposizione una vasta libreria di funzioni e metodi per interagire con I fogli di calcolo. In particolare è possibile estrarre un insieme di celle in uno specifico insieme determinato dagli indici di riga e di colonna attraverso query di tipo *spreadsheet.service.CellQuery*:

```
query = gdata.spreadsheet.service.CellQuery()
query['min-col'] = '1'

start=int(dictionary['start_row'])+index*number_rows_fetched
query['min-row'] = '%d'%start
query['return-empty']='True'
if int(dictionary['end_row'])>=int(query['min-row'])+(number_rows_fetched-1):
        end=int(query['min-row'])+(number_rows_fetched-1)
else:
        end=int(dictionary['end_row'])
query['max-row'] ='%d'%end
feed = gd_client.GetCellsFeed(csv.csv_key.split(':')[1], 1, query=query)
```

Le clausole di estrazione delle celle vengono memorizzate nella variabile dizionario *query*.
I valori contenuti in *query[‘min_col’]*, *query[‘min_row’]*, *query[‘max_row’]* e *query[‘max_col’]* indicano gli indici della prima e dell'ultima cella dell'intervallo da caricare. Mentre *query[‘return-empty’]* indica se devono essere restituite le celle vuote.
Ovviamente per mantenere l'associazione con le classi di rappresentanza specificate nella prima riga è necessario vengano estratte tutte le celle di ogni riga.
L'indice dell'iterazione corrente viene memorizzato nella variabile *index*, *number_rows_fetched* indica invece il numero di righe da caricare ad ogni iterazione.
Procedendo in questo modo è sufficiente specificare l'indice iniziale, ad ogni iterazione l'incremento verrà effettuato correttamente.
Per evitare di sollevare un'eccezione infine deve essere controllato che non venga raggiunta la fine dello spreadsheet. In tal caso l'indice finale nell'iterazione assume il valore dell'indice dell'ultima riga.
Una volta estratte le righe ne vengono analizzati gli elementi contenuti per poter effettuare successivamente la mappatura. A questo proposito viene istanziato un ciclo con un numero di iterazioni fissato al numero di righe caricate. Il codice relativo viene riportato di seguito.

```
processing_row=end-start+1
for count_row in range(0,processing_row):
        #inserimento dei dati relativi alle righe da count_row a count_row+processing_row-1
```

All'interno del ciclo vengono caricati i dati relativi alle entità e importati gli attachment specificati dall'Url associato.
Per quanto riguarda gli attachment, come già citato, GAE non permette la memorizzazione diretta di file esterni. Non è possibile quindi replicare quanto fatto nel capitolo precedente.
La soluzione consiste ancora una volta nell'utilizzo di un servizio fornito da Google per la memorizzazione in un blob del contenuto indicizzato da un URL specifico: *url fetch*.
Le applicazioni App Engine possono comunicare con altre applicazioni o accedere a qualsiasi risorsa generica estraendone il contenuto attraverso il riferimento dato dall'Uniform Resource Locator. Attraverso questo servizo vengono generate richieste http o https e ricevute le relative risposte. La caratteristica principale di questo servizio risiede nell'utilizzo dell'infrastruttura di rete Google garantendo quindi un'alta affidabilità e scalabilità.
La differenza rispetto al procedimento attuato nello step precedente è insita nella memorizzazione degli attachment stessi. Infatti, nella versione precedente, ogni allegato veniva salvato nella cartella relativa all'entità di appartenenza. Nel datastore, invece, veniva semplicemente caricato il percorso del file.
In GAE, non essendo possibile attuare un procedimento di questo tipo, il contenuto dell'allegato viene serializzato e caricato direttamente nel datastore come attributo dell'entry Attachment associata.

```
result = urlfetch.fetch(x[attachment_outfit['fullpath%d'%i]])
```

Successivamente per utilizzare tale allegato, per esempio per la visualizzazione, esso viene inviato attraverso una istanza di HttpResponse al client.

```
response=HttpResponse()
query=db.GqlQuery('SELECT * from Attachment WHERE fullpath=:1','http://www.foto.it/prova.jpeg')
response['Content-type']='image/jpeg'
response.write(query.fetch(1)[0].file)
return response
```

Una volta compiuto l'inserimento nel filesystem dei dati e degli allegati vengono caricate le *n* righe successive ripartendo quindi con l'esecuzione del ciclo.
In primo luogo sembrerebbe che fosse sufficiente solo l'indice relativo alla prima e ultima riga del foglio di calcolo per poter procedere con l'elaborazione dei dati.
Come citato nelle sezioni precedenti, invece, la latenza massima di risposta da parte del server è fissata a trenta secondi. Per questo motivo non è possibile utilizzare un unico processo per l'elaborazione dei dati.
A tale proposito, una volta inviata la configurazione, il numero di righe viene contato e successivamente vengono assegnati degli indici per la gestione della struttura dello spreadsheet.
Questa procedura è stata eseguita per permettere all'applicazione generatrice di elaborare dati provenienti da spreadsheet voluminosi in modo parallelo attraverso il multitasking.
Il servizio offerto da App Engine per l'esecuzione parallela dei processi permette di avviare un insieme di task che continuino la loro elaborazione off-line. Ovvero mentre la risposta può essere servita successivamente alla creazione dei task, il risultato viene restituito in un secondo tempo, al termine dell'esecuzione di ciascuno di essi.
Purtroppo per esigenze di sicurezza GAE non permette l'implementazione multitasking utilizzando le classi fornite da python ma rende obbligatorio l'utilizzo di task implementati secondo le proprie regole e vincoli.

A questo scopo viene fornito il servizio *TaskQueue*, che permette di creare nuovi task e inserirli all'interno di una coda.
Ogni task viene implementato come una vista, ovvero un oggetto che riceve in ingresso una richiesta effettuata e restituisce una risposta.
Dato il numero di righe dello spreadsheet viene calcolato il numero di task necessari. Nel caso questo numero sia uguale a zero è sufficiente un unico processo. Altrimenti vengono creati i processi necessari per l'elaborazione dei dati. Successivamente per ciascuno di essi vengono memorizzati gli indici di inizio e fine riga dello spreadsheet da analizzare.

Il codice relativo al calcolo del numero dei task e alla gestione del caso in cui sia necessario un unico processo viene riportato di seguito:

```
task_number=round(float(row_number)/float(number_rows_per_task))

if int(task_number)==0:
        dict=request.POST
        dict['start_row']=2 #la riga uno contiene gli elementi della prima riga csv e quindi gli indici
        dict['end_row']=row_number+1
        dict['user']=request.META['REMOTE_ADDR']
        taskqueue.add(url='/Task/', params=dict,queue_name='dataqueue')
```

Nel caso sia necessario un numero di task maggiore viene assegnato ad ognuno di essi un insieme di righe attraverso l'attribuzione della coppia di indici relativi alla riga iniziale *start_row* e finale *end_row* assegnate:

```
for i in range(0,int(task_number)):
        dict=request.POST

        if i!=0:
                dict['start_row']=i*number_rows_per_task+1
        else:
                dict['start_row']=2

        if i!=task_number-1:
                dict['end_row']=i*number_rows_per_task+number_rows_per_task
        else:
                dict['end_row']=row_number+1

        dict['user']=request.META['REMOTE_ADDR']
        taskqueue.add(url='/Task/', params=dict,queue_name='dataqueue')

return HttpResponseRedirect('/display_datastore/')
```

L'istruzione *taskqueue.add* è relativa all'inserimento dei task in coda. Ognuno di essi è relativo ad un'unità di lavoro indipendente ed eseguita off-line.
In particolare la struttura di ogni unità di lavoro atomica è costituita da:

- payload contenente dati e parametri
- riferimento all'Url corrispondente alla vista contenente il codice da eseguire

Una volta creato il task e inserito nella coda per l'elaborazione App Engine lo esegue il prima possible, ovvero appena vengono liberate dagli altri processi risorse sufficienti, a meno che non venga specificato un particolare criterio di scheduling. Per quanto riguarda la sincronizzazione e l'accesso condiviso alle risorse non è possibile specificare o istanziare alcun metodo in quanto interamente gestito dalla piattaforma distribuita.

Il tempo massimo di attesa è fissato a dieci minuti. Se un task non viene fatto partire entro tale intervallo dalla sua creazione viene sollevata un'eccezione.
Il problema principale di questo procedimento, che è ancora in fase sperimentale essendo stato rilasciato con l'ultima versione di GAE, è insito nella durata massima di elaborazione del singolo processo, che attualmente è di trenta secondi. Se l'esecuzione non è stata terminata il task fallisce e viene posto nuovamente in coda per essere fatto partire nuovamente da zero. In questo modo viene sprecato molto tempo in quanto tutto il lavoro eseguito viene perso nel caso si tratti del caricamento o dell'importazione di contenuti da server esterni.
Per quanto riguarda le operazioni interne invece è necessario controllare che l'esecuzione di ogni processo sia idempotente. Infatti nel caso venga effettuata l'inizializzazione vengono eseguite nuovamente le stesse operazioni. È necessario assicurare che queste non producano quindi risultati inconsistenti.
Come specificato nella struttura deve essere inserita l'indicazione dell'url relativo al codice da eseguire. È stata quindi realizzata una vista specifica che richiama il metodo di inserimento dati.

```
def Task(request):
   addData(request.POST)
   return HttpResponse()
```

In questo modo i processi vengono eseguiti in background. Una volta terminata la configurazione, infatti, viene visualizzata la pagina relativa al risultato parziale.
Ricaricando tale pagina i dati inseriti vengono aggiornati.
Per poter determinare la fine dell'elaborazione è stato in primo luogo pensato di ricavare il numero di task running dalla stessa task queue. Purtroppo essa non fornisce questo tipo di informazione. Per questo motivo è stato utilizzato nuovamente il database e creata l'entità *Mutex*.

```
class Mutex(db.Model):
   counter=db.IntegerProperty()
   time=db.TimeProperty()
   user=db.StringProperty()
   def __unicode__(self):
      return self.id
```

Il campo *counter* viene inizializzato con il numero di task totali.

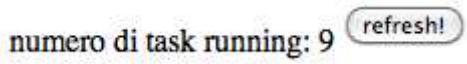

Figura 29 etichetta relativa al numero di task in esecuzione

L'ultima istruzione di ogni processo decrementa il contatore in modo tale l'azzeramento di questo corrisponda alla fine dell'esecuzione multitasking. A questo punto può essere ricaricata la pagina e visualizzato il contenuto finale.

4.2.10 Deployment su Google App Engine

La prima implementazione del lavoro è stata testata sulla macchina locale utilizzando l'SDK di GAE. Per effettuare il porting verso il cloud Google vengono forniti tutti gli strumenti necessari dopo aver registrato la propria applicazione.
Gli strumenti messi a disposizione sono l'importatore per il caricamento delle classi e la dashboard per la visualizzazione delle prestazioni, dell'utilizzo delle risorse e del datastore.
L'applicazione è stata registrata ed è attualmente reperibile all'url

http://hwebappgenerator.appspot.com/index/.

Di seguito vengono riportati i grafici delle prestazioni ottenuti nell'esecuzione dell'applicazione generatrice.
Essi sono relativi all'inserimento nell'ordine(ogni picco è determinato da un'elaborazione distinta e indipendente) di quattro spreadsheet singoli contenenti rispettivamente 200, 400, 750 e 1000 righe e configurati mappandone due campi come attributi di Item e sei come tag senza relazione parent.
I restanti sette esperimenti sono stati compiuti utilizzando uno spreadsheet da 2000 righe configurato le prime cinque volte con due campi mappati come Item e sei come Tag.
Le ultime due configurazioni compiute hanno determinato l'inserimento di dodici campi come Tag e due come Item.

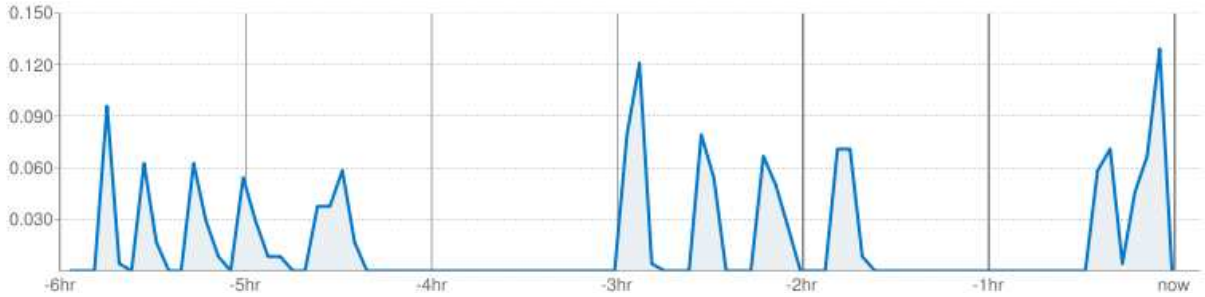

Figura 30 Request/second

Le richieste inviate sono relative ai task inizializzati. Già da questo primo grafico è possibile notare la discontinuità di comportamento della piattaforma distribuita. L'ultimo picco del primo gruppo e i successivi quattro del secondo sono, infatti, relativi alla stessa elaborazione ma, per motivi che verranno illustrati successivamente, variano di una certa quantità.

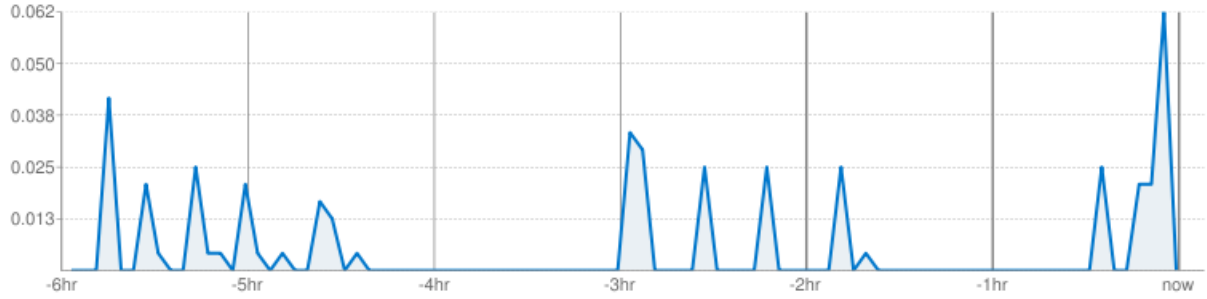

Figura 31 errors/second

L'analisi sugli errori di ogni singola elaborazione mette in luce le caratteristiche del grafico precedente. Infatti task che dovevano produrre lo stesso risultato dato il medesimo input sono stati soggetto di errori distinti e indipendenti probabilmente dovuti all'importazione delle singole righe da Google Docs. Per questo motivo il loro tempo di elaborazione massimo è scaduto e sono stati fatti ripartire nuovamente da App Engine sprecando quindi un grande numero di risorse dovute principalmente al caricamento delle righe duplicate.
Essendo tuttavia l'esecuzione idempotente i dati inseriti non vengono modificati quindi il numero di operazioni nel datastore che vengono effettuate è limitato. Per ogni set di elementi infatti ne viene prima controllata la presenza e successivamente avviene l'eventuale inserimento.

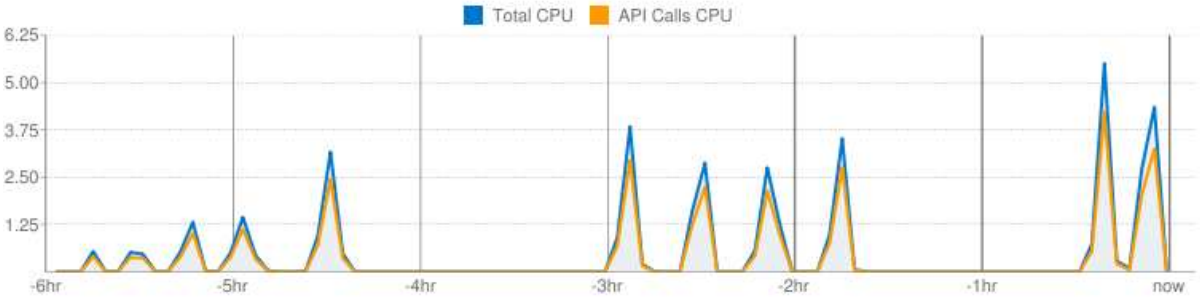

Figura 32 CPU used/second

Per quanto riguarda la quantità di risorsa CPU utilizzata il grafico fornito valida ulteriormente quanto illustrato per gli altri. È possibile notare infatti come il tempo di elaborazione cresca linearmente con la dimensione dello spreadsheet in input variando limitatamente per quanto riguarda le stesse elaborazioni compiute sugli stessi input e quindi derivate dalle ripartenze dei task.

Rispetto alla versione locale ovviamente l'esecuzione nella piattaforma distribuita è molto più veloce. Per quanto riguarda la memorizzazione dei dati nel datastore distribuito a differenza della versione locale gli id assegnati sono numeri molto elevati.

ITEM

ITEM ID	code	description	tags	IP
1050092	WPelle 1	portafoglio pelle 1	[1157096L, 1107102L, 1178001L]	80.180.112.25
1054105	MLana 1	maglione lana 1	[1157096L]	80.180.112.25
1056112	GLana 1	golf lana 1	[1157096L]	80.180.112.25
1058107	PJeans 1	pantalone jeans 1	[1157096L]	80.180.112.25
1075104	Gpelle 4	giaccone di pelle	[1048106L, 1057084L]	80.180.112.25
1077100	GPelle 2	giaccone di pelle 2 HD	[1157096L, 1078097L]	80.180.112.25
1088101	WPelle 2	portafoglio pelle 2	[1037108L, 1126087L]	80.180.112.25
1091114	SLana 1	sciarpa lana 1	[1092096L]	80.180.112.25
1105105	CLana 1	cappello lana 1	[1157096L]	80.180.112.25
1107103	PJeans 4	pantalone jeans 5	[1150111L]	80.180.112.25
1109124	PJeans 2	pantalone jeans 2	[1048106L]	80.180.112.25
1117092	PJeans 3	pantalone jeans 3	[1048106L]	80.180.112.25
1120110	GPelle 1	giaccone di pelle 1	[1037108L, 1173094L]	80.180.112.25
1123093	PJeans 4	pantalone jeans 4	[1048106L]	80.180.112.25
1150110	GPelle 2	giaccone di pelle 2 D	[1126086L, 1124104L]	80.180.112.25

Figura 33 esempio di output dell'applicazione generatrice per le entità Item e Tag

Item contiene il set di elementi dello spreadsheet caricato. Nella tabella(fig. 33) non sono state riportate le colonne di *model_code* e *model_description* in quanto nessun campo è stato mappato in questi attributi.

TAG

TAG ID	type	name	PARENT ID	IP
1037108	colore	marrone	None	80.180.112.25
1048106	colore	grigio	None	80.180.112.25
1057084	acrilico	10	1048106	80.180.112.25
1078097	acrilico	10	1157096	80.180.112.25
1092096	colore	grigio, beige	None	80.180.112.25
1107102	acrilico	30	1157096	80.180.112.25
1124104	acrilico	27	1126086	80.180.112.25
1126086	colore	nero, bianco	None	80.180.112.25
1126087	acrilico	30	1037108	80.180.112.25
1150111	colore	blu	None	80.180.112.25
1157096	colore	nero	None	80.180.112.25
1173094	acrilico	80	1037108	80.180.112.25
1178001	acrilico	40	1157096	80.180.112.25

Figura 34 esempio di output dell'applicazione generatrice per l'entità Tag

La configurazione relativa alla tabella(fig. 34) è stata compiuta mappando le classi ‘colore’ e ‘acrilico’ come tag inserendo inoltre la clasusola parent del secondo rispetto al primo.
Il fatto che siano presenti simultaneamente molte applicazioni che utilizzano il servizio condividendone il datastore implica infatti un numero elevato di entry presenti e soprattutto una differenza sensibile di id anche tra istanze inserite contemporaneamente dall’applicazione generatrice.

4.3 Conclusioni

Rispetto alla versione centralizzata i risultati ottenuti sono molto meno costanti. La discontinuità si manifesta soprattutto nell’analisi delle prestazioni. Lo stesso spreadsheet può impiegare in momenti diversi un tempo sensibilmente maggiore. A prima vista sembrerebbe un’inefficienza introdotta dalla struttura distribuita.
In realtà è necessario considerare alcuni fattori. In primo luogo il server utilizzato nello step precedente era installato nella macchina locale e quindi dedicato interamente all’applicazione generatrice mentre App Engine è un servizio aperto e utilizzato da milioni di applicazioni contemporaneamente. Inoltre le prove effettuate per Apache sono state realizzate caricando lo spreadsheet da locale interamente nel filesystem del server, mentre in GAE i dati vengono caricati row-based solo da GDocs, aggiungendo quindi l’overhead dovuto all’autenticazione e all’invio delle richieste multiple per ottenere lo stesso insieme di dati. Infine deve essere considerato che, mentre Apache è un sistema maturo App Engine è ancora alla prima versione.
In particolare per l’insieme delle API e dei servizi offerti rappresenta attualmente un’ottima scelta per lo sviluppo di un portale di e-commerce dato il datastore disponibile e la possibilità di gestione e di editing delle foto caricate.
Se l’importazione dei dati è risultata molto più lenta del previsto(probabilmente per il controllo sul token di autenticazione dato che il server di importazione è parte del cloud Google), il caricamento degli allegati è molto più veloce rispetto alla versione centralizzata. Questo perché Google all’interno dei propri server ospita miliardi di pagine e l’importazione dei contenuti è sicuramente molto più veloce rispetto a qualsiasi altro servizio.

Per quanto riguarda il processing dei dati per i motivi citati non è stato considerato comparabile il confronto con la versione centralizzata. Un'ipotesi maturata dall'analisi delle prestazioni è che l'elaborazione e la memorizzazione dei dati sia molto più veloce rispetto alla versione centralizzata. Mentre i vincoli relativi all'importing dei dati e alla durata dei processi rendono di fatto il tempo di esecuzione maggiore per quanto riguarda il processo generatore.
Partendo da questo fatto le future release dovranno essere focalizzate nel migliorare l'efficienza dell'importazione utilizzando gli strumenti che verranno rilasciati nelle versioni successive di GAE.

A questo punto l'applicazione è stata implementata nella struttura distribuita.
Dopo aver realizzato la base di dati Global e implementato l'applicazione generatrice nelle due strutture di calcolo illustrate la parte back-end è conclusa.
Come illustrato nell'introduzione e nelle sezioni precedenti la generazione di nuove applicazioni corrisponde a quella di nuovi cataloghi.
Per dimostrare le potenzialità dell'utilizzo di queste nel prossimo step viene progettata e implementata un'applicazione front-end che permetta di visualizzare il risultato di configurazione su dispositivo IPad.

Capitolo 5

Client iPad di testing delle applicazioni generate

5.1 Background

5.1.1 Cocoa e Objective C

Il framework Cocoa[16] consiste in un insieme di librerie, API e programmi eseguibili che formano il livello di supporto per la programmazione e lo sviluppo di applicazioni in ambiente MAC OS X. Attraverso l'utilizzo del framework è possibile, infatti, creare programmi utilizzando gli strumenti e gli oggetti grafici propri di Apple, interagire con i sensori ed eseguire operazioni a basso livello avendo l'accesso al sistema operativo Unix sottostante.
La maggior parte di Cocoa è implementata in Objective-C, un linguaggio orientato agli oggetti compilato e derivato dal linguaggio C.
La caratteristica principale di Obj-C risiede nel fatto che, oltre ad essere compilato per garantire un'esecuzione efficiente, viene dinamicamente interpretato per assicurare una notevole flessibilità. Al momento del lancio dell'applicazione infatti, l'interprete istanzia oggetti basandosi sulla logica di esecuzione e non soltanto sulla compilazione effettuata.
Un programma in esecuzione può, una volta che l'utente genera un evento, caricare un'interfaccia, connettere gli oggetti grafici Cocoa al codice e lanciare il metodo specificato, senza necessità di effettuare nuovamente la compilazione.
Cocoa utilizza la struttura descritta nel corso del terzo capitolo: Model-View-Controller.
Come per il framework django la parte relativa a Model gestisce l'accesso e la gestione al database relazionale attraverso la specifica del modello logico, View le viste per la visualizzazione dei modelli e la manipolazione degli eventi, Controller infine attiva le parti interessate nel corso dell'esecuzione costituendo quindi la parte logica del framework.
Per lo sviluppo delle applicazioni Cocoa viene fornita dalla stessa Apple una suite dedicata: Xcode. Essa è composta da un IDE e dal programma Interface Builder. Il primo viene utilizzato per lo sviluppo del codice, il secondo per la creazione delle interfacce utente.
Utilizzare il modello MVC significa rendere indipendenti le procedure fondamentali di creazione dell'applicazione. È possibile infatti generare le interfacce utilizzando Interface Builder sviluppando nell'IDE i controllori associati mappandone i riferimenti in modo veloce ed immediato.
Per quanto riguarda la compatibilità sono disponibili gli SDK per la programmazione iOS 4.2.x su dispositivi mobile(tra i quali IPad che verrà illustrato nella sezione successiva).
Cocoa include i framework *AppKit* e *Core Foundation* che forniscono strutture comuni per la progettazione delle applicazioni Mac. Le caratteristiche dei framework citati vanno da un set di API di alto livello per creare e gestire effetti particolari con poche linee di codice fino alla possibilità di manipolare qualsiasi aspetto del core.

L'utilizzo di questi strumenti e del supporto della Virtual Machine emulatrice hanno permesso la creazione del programma client-side per la sincronizzazione e il recupero dei dati e metadati dall'applicazione generatrice sviluppata nei punti precedenti.

5.1.2 IPad

IPad[17] è un computer *tablet* progettato, sviluppato e commercializzato da Apple come piattaforma per contenuti audio-video come testi, filmati, file musicali e pagine web.
Il dispositivo è stato rilasciato nella sua prima versione da Apple nell'Aprile 2010 e negli ottanta giorni successivi ne sono stati venduti più di tre milioni.

In comune con i dispositivi Apple IPhone e IPod touch condivide uno schermo totalmente multi touch di 9.7 pollici. Ed è proprio la caratteristica di essere totalmente multi touching ad averne decretato il successo in quanto ha rappresentato una vera e propria innovazione nel mercato dei tablet al tempo del suo rilascio.
Di seguito vengono elencate le caratteristiche tecniche più importanti:

- Schermo da 9.7 pollici multi touch con risoluzione 1024x768 pixel a 132 dpi
- Unità di memoria flash da 16, 32 o 64 GB.
- Connettività Wi-fi 802.11(a/b/g/n), UMTS/HSDPA, GSM/EDGE , Bluetooth
- Processore System-on-a-Chip Apple A4 a 1 Ghz
- Sensore di luce ambientale e accelerometro
- Sistema operativo iOS 4.2.2(la prima versione comprendeva iOS 3.2)

Per la gestione del dispositivo e l'installazione di nuove applicazioni è necessaria la sincronizzazione con il programma ITunes attraverso connessione USB con un personal computer.
Oltre al sensore relativo al multi touch sono gestibili e attivabili eventi relativi ad altre azioni registrabili dai sensori riportati.
Il sensore di luce in base alla luminosità presente aumenta o diminuisce il contrasto e quindi la visibilità dello schermo. L'accelerometro invece viene utilizzato per monitorare l'orientamento dello schermo e passare da una visualizzazione all'altra mantenendo la stessa vista. A differenza dell'IPhone le applicazioni IPad fornite con il dispositivo forniscono pieno supporto per la rotazione dello schermo con la specifica di quattro stati possibili.
Come citato nella parte precedente viene fornito dalla stessa casa produttrice un software development kit condiviso con IPhone per lo sviluppo delle applicazioni in ambiente iOS 4.2.
Il grande successo nella vendita di questo dispositivo lo ha reso una scelta sempre più frequente per quanto riguarda soprattutto il mondo del commercio e del marketing, ambito nel quale è stata sviluppata la tesi compatibilmente con il mercato aziendale.

5.2 Implementazione

5.2.1 Strumenti utilizzati

Anche per l'implementazione dell'ultimo step è stata utilizzata la macchina Apple citata nella sezione relativa nel capitolo tre.
Per quanto riguarda la programmazione sono stati installati Xcode v3.2.5 e la macchina virtuale e l'SDK per iOS 4.2.1.
Oltre alle librerie Cocoa è stato utilizzato un ulteriore framework sviluppato all'interno di un progetto di ricerca e sviluppo aziendale del quale fa parte il lavoro di tesi compiuto: *Nuxie*. Esso comprende librerie per la gestione della base di dati e degli allegati associati.
Il linguaggio di programmazione utilizzato è, come citato nella prima sezione di questo capitolo, Obj-C.

5.2.2 Requisiti e obiettivi

Dopo aver implementato nelle strutture centralizzata e distribuita l'applicazione generatrice è stato richiesto un semplice client per verificare la sua corretta esecuzione. Essendo la capacità di sviluppare applicazioni in ambito iOS uno dei punti forti dell'azienda tale client deve essere progettato e ottimizzato per la visualizzazione nel dispositivo iPad.
A tale scopo una volta effettuata la configurazione deve essere possibile nell'applicazione client-side scaricare l'insieme strutturato dei dati e dei metadati da Google App Engine o Apache in un formato adatto alla rappresentazione di questi nel dispositivo attraverso l'utilizzo di viste e tabelle distinte.

Inoltre, tutte le informazioni, compresi gli allegati e i tag, devono essere visibili in modo semplice e ordinato utilizzando gli input forniti dall'utente.

5.2.3 Creazione del modello

Il primo passo consiste nella creazione del modello dei dati front-end. Viene fornito da Cocoa un framework specifico per la gestione della base di dati: *Core Data*.
Nel corso dell'implementazione esso non è stato utilizzato direttamente in quanto, come già citato, all'interno dell'azienda è attualmente in corso un progetto di ricerca e sviluppo di un framework per rendere più semplice e immediata la realizzazione di una generica applicazione.
Per quanto riguarda la base di dati è presente nel framework Nuxie un wrapper di Core Data: *NXLiquidModel*.
La libreria consente di specificare le informazioni relative al modello in dei file appositi. Oltre ad essa viene fornito uno script che, analizzando i file contenenti la rappresentazione del modello logico, procede con la creazione delle classi objective-c relative ai modelli.
Per la creazione del database e il riempimento dello stesso devono essere specificati due file distinti. Uno contenente la descrizione del modello logico e quindi dei *metadati*, il secondo invece relativo alle entry di ogni singola entità, e quindi ai *dati*.

Tutte le informazioni devono essere fornite in una struttura adatta. Per questo motivo è stato utilizzato ancora una volta il formato json. In particolare è stato utilizzato il protocollo aziendale per la specifica delle entità, degli attributi e dei riferimenti esterni.
L'insieme delle entità viene specificato nella lista *entities*. Essa contiene, per ognuna di esse:

- Il nome
- Un parametro booleano che indica se si tratta di un *JoinModel* o meno. Dove con JoinModel si intende un'entità utilizzata per la creazione di un'associazione 'molti a molti'
- La lista degli attributi, composta da nodi contenenenti ciascuno un dizionario che indica il nome dell'attributo associato, il tipo e l'opzionalità nell'inserimento.

Le associazioni vengono gestite invece attraverso la specifica delle chiavi di riferimento all'interno della lista degli attributi delle entità coinvolte.
La realizzazione avviene specificando nell'attributo relativo alla chiave:

- L'entità associata.
- Il tipo di relazione che lo lega a tale entità, essa può essere di tipo *hasMany*(indicante quindi un'associazione con zero o più entry) o *belongsTo*(indicante l'associazione con al massimo una entry).
- La chiave di riferimento dell'entità associata.
- L'attributo dell'entità associata che costituisce la chiave di riferimento inversa.
- L'entità JoinModel nel caso venga realizzata un'associazione 'molti a molti'.

Inoltre, a differenza dei modelli logici specificati per django e App Engine, ampiamente descritti nei capitoli precedenti, il riferimento deve essere specificato in entrambe le entità coinvolte nell'associazione.

Per quanto riguarda i dati da inserire nel database il protocollo per la definizione del file json è associabile strutturalmente a quello dei metadati. L'applicazione client-side deve infatti relazionare ogni insieme di attributi per il caricamento nell'entità appropriata.
L'inserimento delle entry avviene in un dizionario indicizzato con le entità di riferimento. Ad ognuna di queste corrispondono due liste. Una relativa agli oggetti da aggiornare, l'altra a quelli da eliminare. Tuttavia, nel progetto sviluppato, essendo l'inserimento e la visualizzazione le funzioni sviluppate viene sempre utilizzata la lista relativa all'aggiornamento.

In entrambe le liste ogni nodo è costituito da un dizionario avente per chiavi gli attributi specificati nel modello relazionale relativi all'entità associata. Per ognuna di queste viene riportato il valore dell'attributo mappato.
I campi relativi alle chiavi esterne invece contengono gli identificativi delle istanze di riferimento.

Dopo aver illustrato il protocollo di definizione e specifica dei dati e dei metadati richiesti dall'applicazione Nuxie, per la creazione e l'inserimento, nelle sezioni successive viene fornita l'implementazione compiuta per i singoli casi. Ovvero come effettivamente vengono prodotti i file specificati con parti del codice prodotto e una spiegazione dettagliata dei metodi necessari server-side.

5.2.3.1 Creazione del modello: importing metadati

La necessità di importare i dati e quindi di creare il modello relativo per la loro memorizzazione non comporta l'obbligo di replicare la medesima struttura server-side. Infatti per le caratteristiche in termini di capacità di elaborazione molto minori disponibili nel dispositivo mobile talvolta non è possibile importare dei modelli molto complessi.
Per la gestione di questi casi viene solitamente implementata client-side una versione denormalizzata del modello relazionale. Successivamente il server fornirà i dati in un formato adatto alla nuova struttura rendendone veloce ed immediato l'utilizzo e diminuendo in questo modo i costi in termini di complessità e prestazioni del dispositivo mobile.

Nel corso di questo progetto è stato discusso, tuttavia, come la denormalizzazione delle tabelle sia stata una parte fondamentale per la velocità di esecuzione delle operazioni di ricerca, l'adattabilità a BigTable e adesso per la portabilità su un dispositivo mobile con limitate risorse hardware.
Avendo effettuato questa scelta in fase di progettazione non è stato quindi necessario modificare il modello relazionale implementato per la web application generatrice.
Per la sua definizione client-side è quindi sufficiente specificare tale schema in un file json secondo il protocollo descritto in precedenza.
Di seguito viene riportato il codice relativo alla definizione dell'entità Item e di tutti gli attributi statici tranne le chiavi esterne.

```
[…]
{       "entity": "Item",
        "joinModel": false,
        "attributes": [
                {       "name": "server_id",
                        "type": "string",
                        "optional": true},
                {       "name": "code",
                        "type": "string",
                        "optional": true},
                 {      "name": "description",
                        "type": "string",
                        "optional": true},
                {       "name": "model_code",
                        "type": "string",
                        "optional": true},
                 {      "name": "model_description",
                        "type": "string",
                        "optional": true},
[…]
```

Tra gli attributi è possibile notare *server_id*, nel corso della preparazione dei dati il processo di serializzazione delle entità lato server provvederà ad inserire gli id presenti nelle tabelle come valori di questo attributo.
Per quanto riguarda la dichiarazione dei riferimenti esterni viene riportato di seguito il codice relativo alle chiavi dell'entità Item e alle entità associate Item_Outfit e Outfit.

```
[…]
{       "entity": "Item",
        "joinModel": false,
        "attributes": [
                […]
                {       "name": "item_detailss",
                        "type": "hasMany",
                        "optional": true,
                        "destEntity": "Item_details",
                        "lookupKey": "server_id",
                        "deleteRule": "",
                        "inverse": "item_id"},
                {       "name": "outfits",
                        "type": "hasMany",
                        "optional": true,
                        "destEntity": "Outfit",
                        "lookupKey": "server_id",
                        "deleteRule": "",
                        "inverse": "items",
                        "joinModel": "Item_Outfit"},
                {       "name": "tags",
                        "type": "hasMany",
                        "optional": true,
                        "destEntity": "Tag",
                        "lookupKey": "server_id",
                        "deleteRule": "",
                        "inverse": "items",
                        "joinModel":"Tag_Item"},
        […]
        ]
  },
{       "entity": "Item_Outfit",
        "joinModel": true,
        "leftEntity":"Item",
        "rightEntity":"Outfit",
        "leftLookupKey":"item_id",
        "rightLookupKey":"outfit_id",
        "leftAttributeName":"outfits",
        "rightAttributeName": "items",
        "idKey":"id"},
  {     "entity": "Outfit",
        "joinModel": false,
        "attributes": [
                {       "name": "server_id",
                        "type": "string",
                        "optional": true},
                {       "name": "code",
                        "type": "string",
                        "optional": true},
                {       "name": "description",
                        "type": "string",
```

```
                "optional": true},
        {       "name": "items",
                "type": "hasMany",
                "optional": true,
                "destEntity": "Item",
                "lookupKey": "server_id",
                "deleteRule": "",
                "inverse": "outfits",
                "joinModel": "Item_Outfit"},
[…]
```

Gli attributi chiave di Item, a seconda dell'entità a cui sono associati, vengono mappati in maniera distinta:

- *item_detailss* rappresenta l'associazione con l'entità Item_details. È stato specificato il tipo *hasMany* in quanto le due entità sono associate 'uno a molti'. Il valore *item_id* inserito in corrispondenza di *inverse* implica invece la presenza dell'attributo nell'entità Item_details come riferimento ad Item.
- *outfits* è il riferimento alle entry dell'entità Outfit. Ne viene specificato il JoinModel *Item_Outfit* in quanto è necessaria un'entità di supporto per la realizzazione della relazione di tipo 'molti a molti'. È possibile notare inoltre che il valore inserito nel campo *inverse* di Item.outfits corrisponde all'attributo *items* di Outfit.
- *tags* consiste nell'insieme degli oggetti tag relativi all'istanza di Item specifica. Come descritto nell'implementazione dell'applicazione generatrice Tag viene gestita in modo diverso in quanto deve essere collegabile a tutte le entità presenti. È stato quindi creata un'entità JoinModel per ognuna di esse per associare una data entry di tag a qualsiasi altra istanza di ogni tabella.

Il modello specificato in questo modo verrà utilizzato nel caricamento dell'applicazione per la creazione del modello logico. Essendo quest'ultimo statico è sufficiente riportarlo manualemente senza fornire alcun metodo per l'aggiornamento o la ridefinizione.
Non è stato implementato per questo motivo alcun metodo del server per la generazione del codice relativo ai metadati.

5.2.3.2 Creazione del modello: importing dati

A contrario del modello dei metadati, che rimane statico per tutte le esecuzioni dell'applicazione di testing client-side, l'insieme dei dati con i quali viene popolato il database rappresenta l'output prodotto dall'applicazione generatrice. Il file relativo deve essere quindi creato dinamicamente e restituito dal server a seguito di ogni elaborazione.
A questo scopo nella classe definitrice del modello logico nel progetto dell'applicazione eseguita in App Engine sono stati aggiunti, per ogni classe, i metodi per la serializzazione dei suoi attributi. L'insieme di questi viene inserito successivamente in un dizionario che verrà inviato al client IPad attraverso la richiesta effettuata tramite l'url *http://hwebappgenerator.appspot.com/getJson/*.
La vista associata all'indirizzo fornito è la seguente:

```
def getJson(request):
    s=Serializator()
    response=HttpResponse(str(simplejson.dumps(s.get_json_dict(), indent=2, ensure_ascii=False)),
            mimetype='application/json')
    response['Content-Disposition']='attachment; filename= stream_json.json'
    return response
```

L'oggetto Serializator descrive un serializzatore dell'intera base di dati. Esso, attraverso il metodo *get_json_dict(),* restituisce, secondo il protocollo illustrato, il dizionario dei dati che, successivamente, viene convertito nel formato json ed inviato in risposta al client.
A seguito della sincronizzazione tra quest'ultimo e il server sarà possibile il popolamento del database e l'utilizzo dell'applicazione finale generata.
Viene di seguito riportato un'esempio di json relativo ad una determinata entry dell'entità Item inserita all'interno del datastore BigTable:

```
"Item": {
   "deleted": [],
   "updated": [
    {
     "valid_from": null,
     "server_id": "1037301",
     "model_code": null,
     "description": "cappello lana 1",
     "language": null,
     "deleted": null,
     "valid_to": null,
     "last_update": null,
     "note": null,
     "code": "CLana 1",
     "position": null,
     "model_description": null
    }
```

Un'ultima nota deve essere inserita per quanto riguarda gli allegati. Nel file json sono contenuti, per ogni entry dell'entità, i valori degli attributi. Per consentirne il download da parte del client viene fornito un altro url: *http://hwebappgenerator.appspot.com/getAttachment/'Attachment_id'/.*
Attraverso l'invio della richiesta viene eseguita la vista *getAttachment* che restituisce in output l'allegato specificato dall'identificatore univoco *Attachment_id*:

```
def getAttachment(request, attachment_id):
   attachment=Attachment.get_by_id(int(attachment_id))
   response=HttpResponse(attachment.file, mimetype=type)
   response['Content-Disposition']='attachment; filename='+attachment.filename
   return response
```

La vista restituisce in risposta il file allegato contenente l'attachment relativo alla richiesta effettuata.

5.2.4 Struttura dell'applicazione

Una generica applicazione Cocoa segue il modello MVC[18]. Essa, come citato nelle sezioni precedenti, è formata da un set di viste per l'interfaccia utente, un'unità per la manipolazione della base di dati e dei controllori atti alla gestione delle viste e delle operazioni da effettuare nel database.

Avendo utilizzato il framework Nuxie per la creazione del modello logico e la conseguente modellazione del database è sufficiente specificare le interfacce e i relativi controllori.

5.2.4.1 Struttura dell'applicazione: View

Per la visualizzazione dei dati inseriti sono necessarie tre viste distinte relative rispettivamente agli Item, ai dettagli e agli Outfit caricati. Per queste entità deve essere possibile visualizzare i collegamenti e gli allegati scaricati dal server.

Inoltre l'idea iniziale è stata quella di impostare un download manuale da parte dell'utente dell'applicazione IPad dei dati e degli allegati dal server. Per questo motivo il progetto comprende quattro viste distinte.

- NXUploadDataView: Rappresenta la prima interfaccia all'avvio dell'applicazione, consente di visualizzare il catalogo con i dati scaricati precedentemente e memorizzati nel database o di aggiornarlo facendo partire nuovamente il download dal server.
  Inoltre essendo la prima vista caricata deve essere visualizzata l'immagine di interfaccia globale fornita.
- NXPrebrowsingView: Nel caricamento del catalogo è stata compiuta una suddivisione gerarchica degli elementi presenti. Il livello più alto, relativo ai primi oggetti visualizzati, è dedicato all'insieme specifico degli outfit presenti.
- NXGenericbrowsingView: In questa vista vengono caricati, successivamente alla selezione di un outfit dalla precedente, la lista degli item e relativi allegati. Si ricorda a tal proposito che per outfit si intende un insieme eterogeneo di Item.
- NXDetailbrowsingView: Alla selezione di un item specifico viene caricato l'insieme dei dettagli associati. Anche in questo caso deve essere possibile la visualizzazione degli allegati.

Per l'implementazione è stata creata l'interfaccia grafica associando successivamente un controllore dedicato. La funzione di quest'ultimo è quella di utilizzare gli strumenti generici della vista istanziandone il contenuto in base all'elaborazione prodotta e agli eventi registrati.
Ogni view è infatti un raccoglitore di oggetti creata utilizzando il programma Interface Builder.
Nell'Ide Xcode invece viene implementata la classe ***View****Controller* associata nella quale vengono specificati i metodi e le procedure gestionali. Nella classe possono essere implementate per esempio le funzioni relative alla pressione di una certa area dello schermo o le operazioni da compiere al momento del caricamento e della chiusura.
La prima vista è formata da un'immagine di interfaccia caricata dal server e due oggetti di tipo *UIButton* visualizzati come due pulsanti.

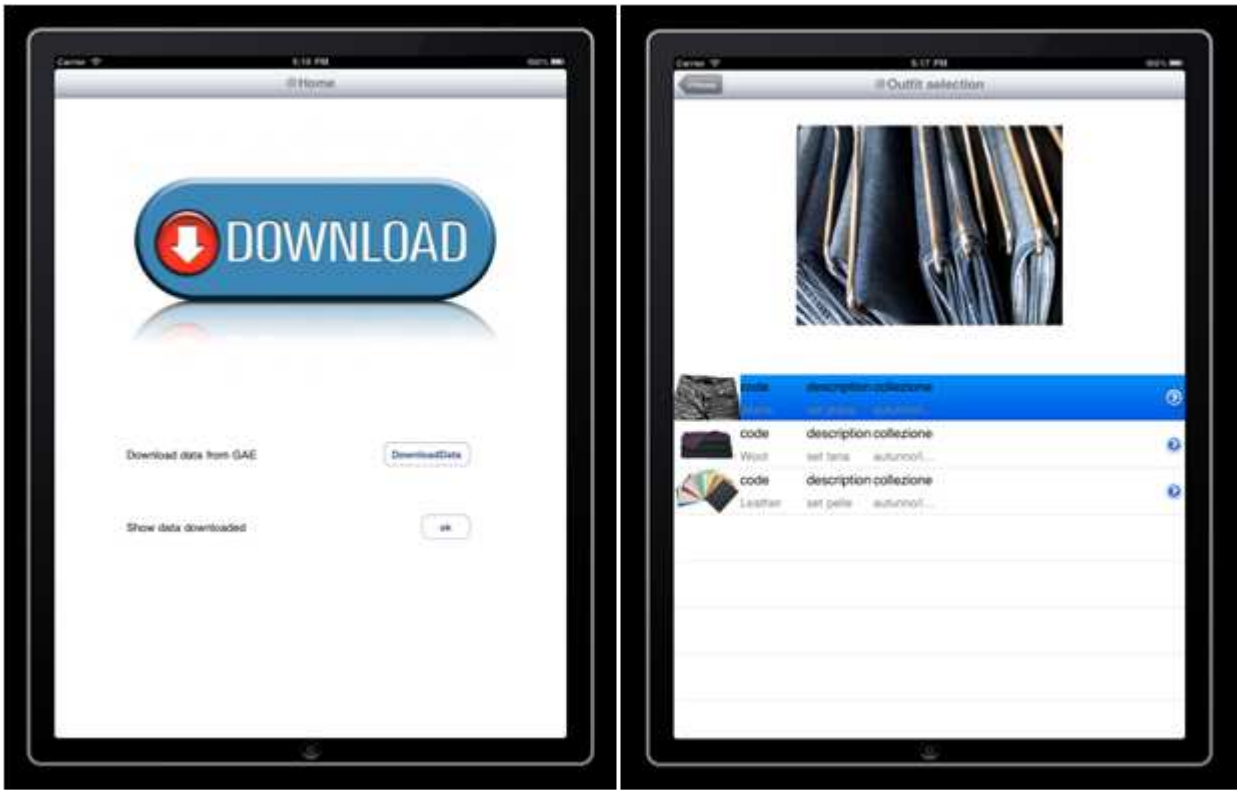

Figura 35 screenshot relativi alle viste di Upload e PreBrowsing

Il primo pulsante, della vista di caricamento dati(fig. 35 a sinistra), consente di scaricare i dati stessi dal server, il secondo di visualizzare il catalogo non aggiornato. Devono essere quindi definite due azioni distinte dipendenti dall'evento generato dall'utente.
A questo scopo sono stati implementati nella classe relativa al controller due metodi indipendenti: *download Data* e *goToCatalogue*.
Per mappare l'evento di selezione, del pulsante apposito, viene specificato il nome del metodo da richiamare direttamente nell'interfaccia di sviluppo utilizzando il programma Interface Builder.

Le altre tre viste hanno una struttura comune e in realtà possono essere definite come un'unica interfaccia dipendente dal contesto.
Essa è formata da un oggetto di tipo *UITableView* relativo ad una tabella le cui righe corrispondono ad istanze del modello relativo al contesto. Inoltre selezionando una qualsiasi riga viene visualizzata la lista degli allegati relativi attraverso l'utilizzo di un'istanza di *UIScrollView*.
Entrambi gli elementi visualizzati sono in realtà delle sotto-viste. Cocoa permette infatti di inserire elementi in qualsiasi vista mappandoli come sotto-viste indipendenti l'una dall'altra e gestibili da un unico controllore.
La prima problematica consiste nell'inserimento dei dati nella tabella fornita. Essa infatti di default supporta un'unica colonna senza la possibilità di estensione. A questo proposito sono stati ridefiniti alcuni metodi attraverso la tecnica dell'overriding responsabili della singola cella visualizzata.
È possibile infatti mappare in un'unica riga etichette differenti eseguendo l'implementazione nel controllore della view e precisamente definendo nuovamente il metodo *cellForRowAtIndexPath*.
Essendo fondamentalmente composta da un'unica colonna la definizione di cella o riga per una UITableView è indifferente.
In particolare ogni cella corrisponde ad una vista caricata a seconda della scrollatura effettuata dall'utente. Successivamente, in base alle righe da visualizzare, ne vengono caricati gli oggetti specifici.
Aggiungendo alla cella delle sotto-viste, ognuna relativa ad una colonna indipendente, è possibile ottenere una matrice di elementi a due dimensioni adatta quindi all'inserimento delle entry relative al database.
Nelle celle delle tabelle impostate nel progetto sono state mappate, a seconda del modello, *n* viste di tipo UILabel, corrispondenti a delle etichette modificabili. Ognuna di esse corrisponde ad un'attributo dell'entità considerata.
In aggiunta ad ogni cella viene caricata un'immagine icona. Il caricamento e la visualizzazione vengono fornite dal framework Nuxie attraverso l'oggetto *imageManager* di tipo *NXImageManager*.
La classe sviluppata consente di scaricare l'allegato mediante l'Url dal server, come illustrato nella sezione relativa all'importazione dei dati, salvando successivamente il file associato nella memoria di massa e caricandone il contenuto nelle view associate.
Nella cella l'immagine viene caricata in una vista di tipo *NXImageView* e aggiunta a quella principale. In questo modo è possibile ottenere delle icone relative ad ogni oggetto caricato.
La differenza tra queste immagini e gli altri allegati è data dal valore di *entity_attribute* della classe Attachment. In questo caso infatti viene aggiunto al nome dell'entità il suffisso '_icon' e memorizzato su tale attributo.
Se la ricerca per le icone non porta a nessun risultato vengono caricate delle immagini di default memorizzate nel server. Essendo queste relative a tutte le entità, come l'immagine dell'interfaccia di update, si distinguono per il valore del campo entity 'ALL'.
Viene di seguito fornita una pagina relativa ai dettagli dell'oggetto nella quale non sono stati mappati allegati e quindi vengono visualizzate le immagini di default.

Figura 36 vista DetailsBrowsing senza allegati associati agli oggetti

La tabella, come già citato, risponde agli eventi di scrollatura generati dall'utente e ricavati dal sensore touchscreen. Ovvero ad ogni operazione di scrolling i dati presenti vengono ricaricati. Purtroppo questo funziona solo per lo scrolling verticale rendendo quindi impossibile mappare in questo modo i tag.
La numerosa quantità di tag presenti per riga infatti non è in primo luogo prevedibile in quanto sono attributi dinamicamente inseriti e, secondariamente, il loro inserimento non può essere limitato per i requisti dell'applicazione generatrice.
Anche questo problema, come per gli allegati, è stato risolto utilizzando una UIScrollView. Essa è relativa ad una sotto-vista nella quale è possibile creare un insieme illimitato di oggetti di tipo UIView i quali verranno caricati dopo ogni operazione di scrollatura verticale o orizzontale effettuata dall'utente.
Ogni tag, rappresentato dalla coppia <type, value, parent>, viene caricato in una coppia di etichette dedicate. Successivamente queste vengono aggiunte alla vista principale di scrolling. Ovviamente, per questi inserimenti, devono essere specificati i parametri geometrici in modo tale le coppie inserite possano essere visualizzate correttamente ad ogni azione eseguita.
Per quanto riguarda, infine, gli allegati, essi vengono visualizzati alla selezione di una riga della tabella (il trasferimento alla lista degli elementi associati è gestito invece dalla pressione del tasto freccia contenuto alla fine di ogni riga).
Gli allegati vengono visualizzati, come già citato, in una vista di scrolling principale. Essi vengono mappati esattamente come le icone della tabella utilizzando il gestore delle immagini e delle view dedicate. La differenza viene dal contesto della mappatura stessa. In questo caso infatti gli *m* allegati specifici di una determinata istanza di Item, Outfit o Item_details vengono inseriti nella vista di scrolling e visualizzati uno alla volta. Nonostante questa caratteristica il procedimento per inserire gli oggetti avviene nello stesso modo.
Di seguito vengono riportati alcuni screenshot relativi agli oggetti di due Outfit inventati 'pelle' e 'lana'. Il primo composto da oggetti in pelle, il secondo da prodotti in lana.

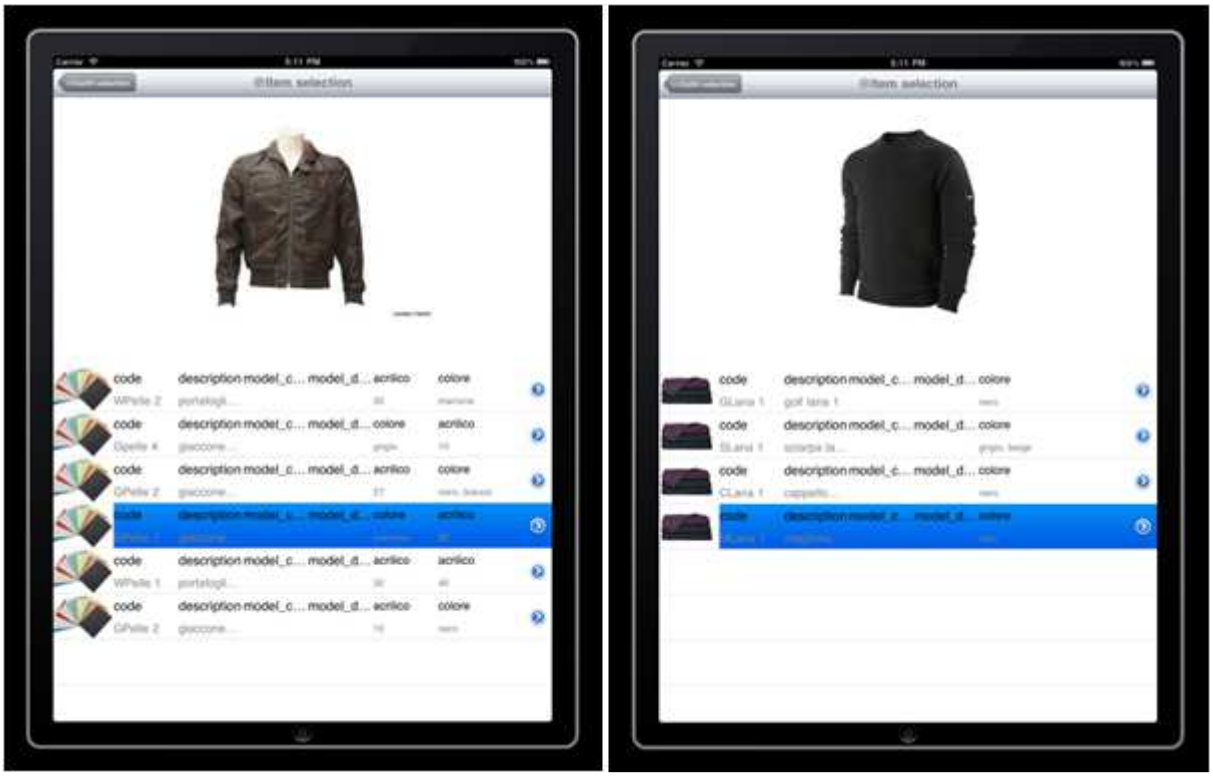

Figura 37 sreenshot relativi a GenericBrowsingView per gli outfit pelle e lana

Come è possibile notare sono stati inseriti come attributi tag nel corso della configurazione la percentuale di acrilico e il colore. Mentre il primo è relativo solo ai capi di pelle il secondo è globale per tutti i prodotti.
L'importanza nell'utilizzo degli attributi dinamici viene evidenziata anche in questo caso. La definizione di un attributo fisso per ogni prodotto avrebbe portato ad una colonna totalmente inutilizzata nel caso di prodotti non in pelle.

5.2.4.2 Struttura dell'applicazione: inizializzazione e navigazione

Nel progetto Cocoa la classe principale responsabile della gestione dell'installazione e dell'esecuzione del programma implementato è *AppDelegate*.
In essa vengono allocati gli oggetti relativi all'intera applicazione, essi sono:

- Model, oggetto di tipo NXGlobalLiquidModel. Nel costruttore dell'oggetto sono definite le procedure per creare le entità e le associazioni tra queste utilizzando il json relativo ai metadati. Successivamente vengono scaricate dal server le entry che verranno inserite nella base di dati.
- Window, oggetto di tipo UIWindow. Rappresenta la finestra nella quale vengono caricate le viste successive. In essa vengono inseriti gli oggetti generici utilizzati nel corso dell'esecuzione come il controller di navigazione.
- Navigation controller, oggetto di tipo UINavigationController. Consente di gestire le viste e i parametri da trasferire tra queste.
- Image manager, oggetto di tipo NXImageManager. Permette il caricamento e l'allocazione di immagini in un'istanza di NXImageView.

Il primo oggetto elencato viene utilizzato per ottenere le istanze di una determinata entità secondo le clausole inserite in un predicato specifico. In questo modo è possibile definire delle query ad alto livello utilizzando il supporto ORM fornito da Cocoa.

Vengono di seguito riportati alcuni esempi di ricerche nella base di dati implementate nel progetto.

```
NSPredicate pred=[NSPredicate predicateWithFormat: @"aserver_id==%@", selectedOutfit];
NSArray *outfits = [[self model] fetchEntity:@"Outfit" predicate:pred sortDescriptors:nil faults:nil];
```

La variabile *pred* definisce la ricerca delle entry aventi l'attributo *aserver_id* pari a *selectedOutfit*(variabile contenente l'id dell'oggetto selezionato nella prima vista).
La seconda istruzione carica in un array tutte le istanze di Outfit compatibili con il predicato descritto. Ovvero tutte le istanze con id pari alla variabile specificata. Data l'univocità dell'identificatore ovviamente in questo caso l'array sarà costituito al massimo da un elemento.

```
pred=[NSPredicate predicateWithFormat: @"(aattachment_entity_id==%@) AND
(aattachment_attribute='item_icon')", ((NXCDItem *)[self.listOfItems
objectAtIndex:indexPath.row]).aserver_id];
NSArray *attachments=[[self model] fetchEntity:@"Attachment" predicate:pred sortDescriptors:nil faults:nil];
```

In questo caso vengono caricati nell'array *attachments* gli allegati di tipo icona relativi all'entry con l'id specificato dell'entità Item.

Nell'applicazione sviluppata l'utilizzo del navigation controller è fondamentale in quanto come descritto nella sezione precedente, da ogni vista il passaggio alla successiva è determinato dalla scelta effettuata dall'utente. Per il caricamento di ogni view distinta deve essere in primo luogo allocato un oggetto controller associato. Questo fatto non costituisce alcun problema in quanto, come è stato ampiamente illustrato nella sezione predente, ognuna di esse è associata ad un controller. Successivamente l'istanza viene caricata in una pila gestita dal navigation controller utilizzando il metodo *pushViewController* come è possibile verificare nel codice riportato di seguente relativo al passaggio dalla vista degli Outfit a quella degli Item associati.

```
NXGlobalGenericbrowsingViewController *dvController = [[NXGlobalGenericbrowsingViewController alloc]
initWithNibName:@"NXGlobalGenericbrowsingView" bundle:nil];
dvController.selectedOutfit = selectedOutfit;
[self.navigationController pushViewController:dvController animated:YES];
```

Il passaggio dei parametri viene realizzato strutturando i controllori delle singole viste con delle variabili di esemplare dedicate, nel codice riportato viene utilizzata la variabile *selectedOutfit* relativa al controller della vista Item.
Una volta caricati uno o più controllori nella pila e completata l'esecuzione corrente questa passa al navigation controller. Dalla pila viene estratto il primo controllore secondo la politica FIFO ed eseguita la view associata.
Le interfacce precedenti tuttavia non vengono deallocate automaticamente e, attraverso le funzionalità offerte dal controllore della navigazione è possibile tornare alle liste caricate effettuando scelte differenti e consentendo quindi una visualizzazione semplice e completa.

5.3 Conclusioni

L'ultimo step rappresenta un esempio delle potenzialità dell'applicazione generatrice.
Utilizzando ed eventualmente estendendo un'applicazione sviluppata in questo modo associata al server Apache o a App Engine è possibile ottenere in modo semplice ed efficiente ma soprattutto impostabile dall'utente un catalogo con la struttura desiderata partendo da un insieme di dati contenuti in un foglio di calcolo generico.

Capitolo 6

Conclusioni generali ed estensioni future

L'aspetto innovativo del progetto è rappresentato in primo luogo dalla base di dati sviluppata.
Oltre agli obiettivi di globalità richiesti, lo studio e l'implementazione di una versione denormalizzata, e dei conseguenti vantaggi introdotti da essa, costituiscono degli aspetti totalmente inediti.
L'utilizzo degli attributi dinamici, inoltre, rappresenta un argomento sul quale è possibile sviluppare molta ricerca. Aggiungendo la caratteristica di inserimento dei tag da parte degli utenti, infatti, l'attribuzione di parametri dinamici può diventare una caratteristica fondamentale per i datastore dei portali web.
Per quanto riguarda l'applicazione generatrice, sono attualmente disponibili nel mercato molti sistemi per la migrazione dei dati da un foglio elettronico ad un database generico. La particolarità del progetto sviluppato è relativa alla totale configurabilità, ovvero alla possibilità di generare nuovi cataloghi distinti semplicemente modificando il modulo di mapping.
Per quanto riguarda l'implementazione eseguita, una possibile estensione riguarda l'aggiunta delle entità illustrate nel capitolo secondo. Questo per permettere, oltre al riempimento delle tabelle del database, la produzione vera e propria di un'applicazione iPad completa, adatta quindi per la visualizzazione delle informazioni ma anche per la gestione e l'invio dei dati relativi alle vendite o ai commenti sui capi.
Il tutto ovviamente impostabile dall'utente attraverso la selezione delle caratteristiche gestionali nella form configuratrice.
Lo sviluppo in una struttura distribuita, quale Google App Engine, costituisce un punto di partenza per un progetto di ricerca molto più approfondito. Oltre all'evoluzione della struttura, e al conseguente miglioramento dei servizi offerti all'applicazione generatrice, sono da valutare altri sistemi di cloud computing PaaS per ottenere una comparazione tecnica affidabile. Analisi da effettuare, soprattutto, per le prestazioni offerte nell'importazione dei dati.

Il progetto sviluppato, nel contesto del programma di ricerca e sviluppo dell'azienda, rappresenta una buona base di partenza. Esso è stato strutturato, infatti, per essere facilmente esteso con l'utilizzo di plug-in aggiuntivi.

Capitolo 7

Conclusioni personali

Fino ad ora la mia carriera professionale è stata contraddistinta da numerose esperienze maturate in aziende nella provincia di Treviso e di Padova.
Purtroppo nell'ambito universitario non è sempre possibile amalgamare le nozioni teoriche con attività pratiche allo stesso livello di approfondimento. Questo, secondo la mia opinione, rappresenta un limite per quanto riguarda l'apprendimento e mantiene uno spazio di divisione tra il mondo del lavoro e l'università.
Nel corso dei miei studi ho sempre cercato di inserire delle attività di ricerca e sviluppo in aziende esterne per poter fare delle esperienze in ambiti e contesti diversi.
La tesi di laurea triennale è stata svolta in una delle maggiori aziende in crescita per quanto riguarda il panorama italiano delle startup: M31[19]. Nell'azienda mi sono occupato di un sistema di gestione delle applicazioni per un dispositivo videotelefono voip successivamente commercializzato.
Lavorare con un team giovane e motivato in un bellissimo ambiente mi ha portato a considerare, per la scelta della tesi di laurea magistrale, il maggiore incubatore italiano di startup: H-Farm.
Le motivazioni riguardanti lo sviluppo di applicazioni su dispositivi mobile Apple e su una struttura distribuita hanno fatto ricadere infine la scelta in H-Umus, società interna dell'incubatore.
L'esperienza è stata molto positiva. Oltre alle conoscenze e agli strumenti inediti, dei quali è stata acquisita padronanza, collaborare con un team di persone competenti e contribuire alla produzione di un progetto di ricerca di questo tipo sono stati fattori determinanti per la mia personale crescita professionale.

Ringrazio Enrico Zeffiro, responsabile del settore tecnico di H-Umus, tutore aziendale e ideatore dell'idea iniziale sviluppata in questo progetto e tutto il team dell'azienda per la collaborazione fornita.
Ringrazio inoltre il professor Sergio Congiu, relatore e tutore a livello universitario dello stage compiuto per la disponibilità offerta.

Riccardo Lorenzon

Bibliografia

[1] http://it.fuqua.duke.edu/public/2001PivotTableIntroduction.pdf

[2] G. Sanders and S. Shin. Denormalization effects on performance of RDBMS. In *Proceedings of the 34th Hawaii International Conference on System Sciences – 2001*, IEEE

[3] M. Karma, M. Baillie and F. Crestani. Tag Data and Personalized Information Retrieval. In *SSM '08: Proceeding of the 2008 ACM workshop on Search in social media*, ACM

[4] F. Lodhi and M. Ghazali. Design of a simple and effective object-to-relational mapping technique. In
SAC 07: Proceeding of the 2007 ACM symposium on Applied computing, ACM

[5] http://www.djangobook.com/en/beta/chapter01/

[6] http://docs.djangoproject.com/en/dev/ref/request-response/

[7] http://docs.djangoproject.com/

[8] http://httpd.apache.org/

[9] http://www.sqlite.org

[10] http://docs.python.org/

[11] http://code.google.com/p/django-tagging/

[12] C. Gong, J. Liu, Q. Zhang, H. Chen and Z. GongThe. Characteristics of Cloud Computing. In *2010-39th International Conference on Parallel Processing Workshop*, ACM

[13] http://code.google.com/appengine/docs/whatisgoogleappengine.html

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes and R. E. Grube. Bigtable: A Distributed Storage System for Structured Data, Google Inc.

[15] C.Ordonez, I. Song and C. Garcia Alvarado. Relational versus Non-Relational In *DOLAP '10 Proceedings of the ACM 13th international workshop on Data warehousing and OLAP*, ACM

[16] http://developer.apple.com/technologies/mac/cocoa.html

[17] www.apple.com/ipad/

[18] http://developer.apple.com

[19] http://www.m31.com

[20] http://code.google.com /appengine/docs

[21] http://code.google.com/intl/it-IT/appengine/docs/python/howto/usinggdataservices.html