BookSales UniBG: iterazione 1

Del Prete Giovanni, Ghilardi Nicola, Polver Marco

March 10, 2020

Contents

0.1 Casi d'uso coperti

Di seguito vengono riportati i casi d'uso coperti dal software realizzato nell'iterazione 1:

1. UC1: Registrazione studente
2. UC2: Login studente
3. UC3: Logout
4. UC10: Ricerca annuncio
5. UC11: Ricerca studente

Per l'implementazione di queste funzionalità si sono resi necessari due ulteriori passi:

1. *Realizzazione dei modelli dei dati*: in Django i modelli sono delle classi Python che rappresentano le tabelle che devono essere realizzate all'interno del database.
2. *Realizzazione di uno script Python per il riempimento del database*: questo passo si è reso necessario per poter avere un numero consistente di dati su cui testare le diverse funzionalità dell'applicazione.

0.2 Note sul processo di implementazione

L'obiettivo iniziale era quello di realizzare questo progetto tramite un processo di Test Driven Development puro, tuttavia è stato necessario modificare questo processo permettendoci di testare alcune componenti del software solo dopo la loro implementazione.
Questo va contro i principi del TDD, per cui bisognerebbe prima ideare una serie di test e solo dopo realizzare un codice sufficiente per passarli, ma si è reso necessario per due motivi:

1. *Il nostro progetto è una applicazione web*: il testing delle applicazioni web risulta essere diverso rispetto al testing di applicazioni desktop e a volte i test risultano difficili anche solo da pensare prima di un'implementazione anche solo parziale della funzionalità da testare.

2. *Inesperienza con Django*: per tutti e tre questo progetto ha rappresentato l'occasione di provare Django per la prima volta. Questo framework è stato scelto in quanto particolarmente utilizzato al giorno d'oggi, ma nessuno di noi l'aveva utilizzato prima di questo progetto. Questo ha reso difficile, soprattutto all'inizio, l'applicazione del TDD, in quanto la realizzazione di un test presuppone una buona conoscenza degli strumenti utilizzati. I risultati ottenuti, tuttavia, sono ottimi e anche la nostra confidenza nell'utilizzo di Django è cresciuta notevolmente con il tempo.

0.2.1 Note sui test effettuati

Per la realizzazione dei test sono stati utilizzati due strumenti:

- La classe *TestCase* fornita da Django, utilizzata per l'esecuzione di test strutturali.
- *Selenium WebDriver*, una famosa API che permette di automatizzare l'utilizzo di un qualsiasi browser, molto utile per l'esecuzione di test funzionali.

Test strutturali

Come già è stato detto in precedenza, i test strutturali sono stati effettuati utilizzando la classe TestCase che viene fornita da Django.
L'esecuzione di test all'interno del framework Django risulta particolarmente comoda in quanto permette di utilizzare un vero e proprio "database di test", evitando quindi di sporcare il database normalmnte utilizzato dall'applicazione con dati utili solo in fase di test.
I test strutturali effettuati per le varie funzionalità dell'applicazione verificano in genere i seguenti punti:

- Utilizzo del template HTML corretto in ogni pagina.
- Restituzione dello status code corretto (quasi sempre 200 o 404) a fronte della richiesta di una pagina web con determinati dati.
- Comportamento corretto dell'applicazione a seguito della somministrazione di dati corretti ed errati.

Test funzionali

In uno script Python è stato utilizzato Selenium WebDriver per effettuare test funzionali.
Le differenze tra i test che si possono eseguire con la classe TestCase e con il WebDriver sono notevoli:

- TestCase facilita l'utilizzo di richieste GET e POST. Selenium WebDriver è invece un puro sostituto dell'uomo, pertanto va istruito su quali tasti cliccare, i testi da inserire nelle varie textbox, ecc.
- Selenium WebDriver non fa parte del framework Django, pertanto le azioni compiute da esso andranno a scrivere nuovi dati nel vero database dell'applicazione e non in un database di prova.

I test funzionali sono stati realizzati basandosi sui *casi d'uso*, pertanto non testano una singola funzionalità dell'applicazione ma la successione delle azioni compiute normalmente da un utente con un certo fine.

Copertura

Solo in poche occasioni la copertura dei test sui file Python presenti all'interno dell'applicazione risulta essere del 100%, questo a causa dei seguenti motivi:

- All'interno dei file di un'app Django sono spesso presenti porzioni di codice di default che potrebbero non essere coperte.

- Alcune funzioni utilizzano la libreria "random", la quale rende il comportamento di alcune porzioni di codice imprevedibile; non è quindi possibile avere una copertura del 100% in certe funzioni.

0.3 L'architettura software

Trattandosi di una applicazione web realizzata con Django, l'architettura software è imposta dal framework utilizzato, pertanto sia in questa iterazione che nella successiva l'architettura non è cambiata. Il pattern utilizzato è il MVT già mostrato nell'iterazione 0.
In questa iterazione sono stati sviluppati i 3 componenti di tale pattern.

0.4 Realizzazione dei modelli dei dati

Non tutte le tabelle progettate nella iterazione 0 sono state tradotte in modelli, in quanto solo una parte delle funzionalità del progetto sono state e saranno implementate, rendendo alcune tabelle superflue.

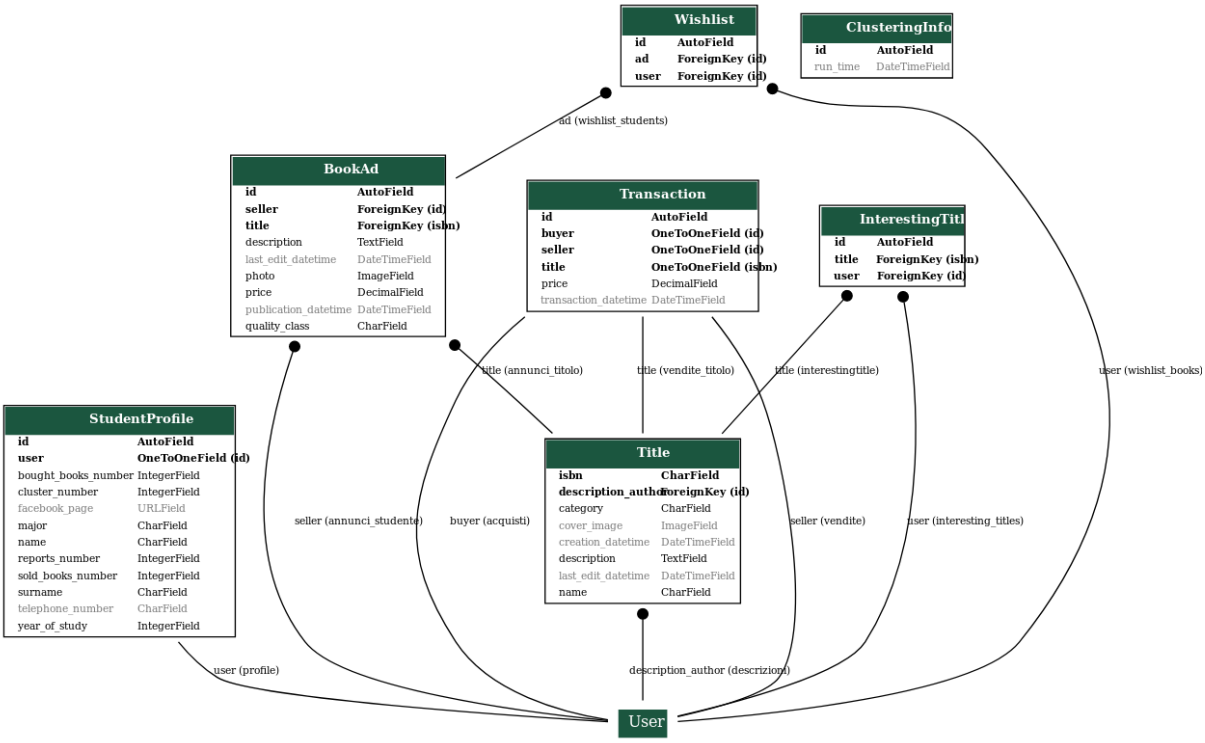

Figure 1: Diagramma dei modelli implementati

I modelli creati sono i seguenti:

- User: modello base fornito di default da Django.
- StudentProfile: modello che si lega a User per fornire ulteriori informazioni sul singolo utente. I suoi campi:
  - user: OneToOneField (simile a Foreign Key) che lega un solo User ad uno StudentProfile.
  - name: nome dello studente.
  - surname: cognome dello studente.
  - telephone_number: numero di telefono dello studente.

  - facebook_page: link alla pagina Facebook dello studente.
  - major: corso di laurea a cui lo studente è iscritto.
  - year_of_study: anno di corso.
  - sold_books_number: numero di libri venduti.
  - bought_books_number: numero di libri acquistati.
  - reports_number: numero di segnalazioni ricevute.
  - cluster_number: numero del cluster cui appartiene.

- Title: modello per i singoli titoli. I campi sono:
  - isbn: ISBN del titolo.
  - name: nome del titolo.
  - cover_image: ImageField per l'immagine della copertina.
  - description: descrizione scritta da un utente.
  - description_author: autore della descrizione.
  - creation_datetime: data di creazione del titolo nel database.
  - last_edit_datetime: data dell'ultima modifica.
  - category: categoria del titolo (fisica, matematica, informatica, meccanica, elettronica, economia, automazione, statistica).

- BookAd: modello che descrive il singolo annuncio per la vendita di un libro. I campi sono:
  - title: ForeignKey che lega un solo Title ad un BookAd.
  - seller: ForeignKey che lega un solo User (il venditore) ad un BookAd.
  - description: descrizione dell'annuncio.
  - price: prezzo del libro.
  - quality_class: classe di qualità del libro (A, B, C, D).
  - publication_datetime: data di pubblicazione dell'annuncio.
  - last_edit_datetime: data dell'ultima modifica dell'annuncio.
  - photo: ImageField per una foto del libro.

- Wishlist: modello che descrive l'appartenenza di un determinato annuncio alla wishlist di un utente. I campi sono:
  - ad: ForeignKey che lega un oggetto di tipo Wishlist ad un solo BookAd, ovvero l'annuncio che fa parte della reale wishlist.
  - user: ForeignKey che lega un oggetto di tipo Wishlist ad un solo utente.

- InterestingTitle: modello che descrive l'appartenenza di un determinato titolo alla lista degli "interesting titles" di un utente. I campi sono:
  - title: ForeignKey che lega un oggetto di tipo InterestingTitle ad un solo Title, ovvero l'annuncio che fa parte della reale lista "Interesting Titles".
  - user: ForeignKey che lega un oggetto di tipo InterestingTitle ad un solo utente.

- CluseringInfo: modello utilizzato per salvare le informazioni sull'esecuzione dell'algoritmo di clustering. I campi sono:
  - run_time: data e ora della specifica esecuzione dell'algoritmo.

I modelli sono stati testati tramite l'utilizzo dello script per il riempimento del database e indirettamente con i test effettuati su altre funzioni del sito web.

0.5 Realizzazione di uno script Python per il riempimento del database

Lo script crea i seguenti oggetti:

- 800 utenti e profili studente.
- 150 titoli.
- Un numero casuale tra 0 e 10 di "interesting titles" per ogni studente.
- Un numero casuale tra 10 e 50 di annunci per ogni titolo.
- Un numero casuale tra 0 e 20 di annunci nella wishlist dei singoli studenti.

0.6 Implementazione delle altre funzionalità

0.6.1 Template

Django mette a disposizione un vero e proprio linguaggio di programmazione utilizzabile all'interno dei template HTML e noi ne abbiamo fatto largo uso.
E' inoltre possibile evitare di ripetere per ogni pagina web porzioni di codice HTML comune, estendendo un template comune contenente proprio quel codice da riutilizzare.

Tutti i nostri template estendono il template "base.html", il quale descrive la barra di navigazione presente in tutto il sito web e la divisione tra titolo e corpo delle singole pagine web.

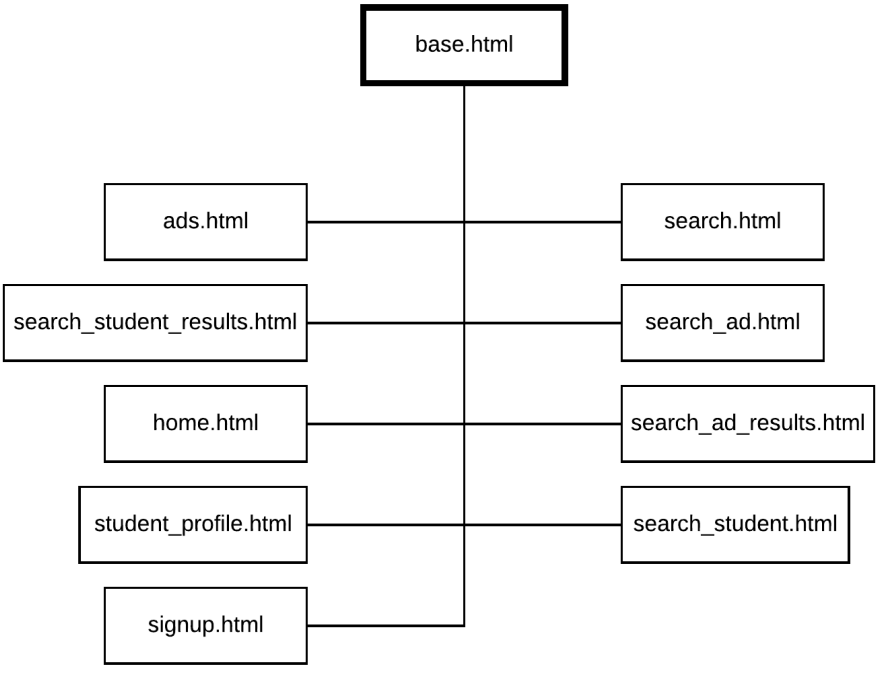

Figure 2: Diagramma dei modelli implementati

0.6.2 Login e logout

Il login e il logout sono stati implementati facilmente, in quanto è stato sufficiente utilizzare quelli già forniti da Django.

0.6.3 Registrazione di un nuovo utente

La registrazione di un nuovo utente implica implicitamente la creazione di un nuovo oggetto User e di un nuovo oggetto StudentProfile. Per questo motivo la pagina per la registrazione presenta due form, uno per la creazione dell'utente e uno per la creazione del profilo studente.
I form sono stati realizzati da noi, così come il controllo sull'indirizzo email inserito (il quale deve appartenere al dominio studenti.unibg.it, coerentemente con le specifiche).
La funzione di registrazione funziona come segue:

Data: request
Result: nuovo User e nuovo StudentProfile salvati nel database
initialization;
if *request.type = 'POST'* **then**
  creazione user_form;
  creazione student_form;
  ricezione dati utente;
  if *user_form valido* ***and*** *student_form valido* **then**
    user ← user_form.data;
    student ← student_form.data;
    student.user ← user;
    return login page ;
  end
else
  user ← user_form.data;
  student ← student_form.data;
end
return signup page

0.6.4 Ricerca studente

Questo caso d'uso richiede l'implementazione di due "view" all'interno della webapp:

- funzione search_users
- classe SearchUsersResults

search_users

Questa funzione lavora nel seguente modo:

Data: request
Result: search_student page
initialization;
create student_search form;
return search_student page with student_search form;

SearchUsersResults

Questa classe contiene un metodo *get_queryset* che ha l'obiettivo di filtrare gli utenti del servizio sulla base dei campi compilati.

Data: username, major, year of study, User objects, StudentProfile objects
Result: list of compatible students
initialization;
users ← {u ∈ User.objects **if** u.username contains username};
profiles ← {p ∈ StudentProfile.objects **if** p.user ∈ users};
object_list ← {o ∈ profiles **if** o.major = major **and** o.year_of_study = year of study};
return object_list;

0.6.5 Ricerca annuncio

Questo caso d'uso richiede l'implementazione di due "view" all'interno della webapp:

- funzione search_ads
- classe SearchAdsResults

search_ads

Questa funzione lavora nel seguente modo:

Data: request
Result: search_ads page
initialization;
create ads_search form;
return search_ads page with ads_search form;

SearchAdsResults

Questa classe contiene un metodo *get_queryset* che ha l'obiettivo di filtrare gli annunci pubblicati sul sito web sulla base dei campi compilati.

Data: title name, quality class, starting price, ending price, Title objects, BookAd objects
Result: list of compatible ads
initialization;
chosen_title $\leftarrow$ {t $\in$ Title.objects **if** t.name = title name};
ads $\leftarrow$ {a $\in$ BookAd.objects **if** a.title = chosen_title **and** a.quality_class = quality class};
object_list $\leftarrow$ {o $\in$ ads **if** starting price $\leq$ a.price $\leq$ ending price};
return object_list;

0.7 Resoconto test (copertura C0)

I test sono stati effettuati sull'intera applicazione, pertanto anche sulle porzioni di codice fornite di default dal framework Django. Queste porzioni di codice non risultano tuttavia interessanti nella valutazione della copertura dei test, quindi valutiamo solo la copertura dei componenti da noi creati.

Coverage for **PycharmProjects/booksales_unibg/sales/views.py** : 52%
169 statements | 88 run | 81 missing | 0 excluded

Figure 3: Copertura C0 delle "view" della webapp, ovvero dei metodi che elaborano i dati

Coverage for **PycharmProjects/booksales_unibg/sales/models.py** : 95%
62 statements | 59 run | 3 missing | 0 excluded

Figure 4: Copertura C0 dei modelli dei dati

Coverage for **PycharmProjects/booksales_unibg/kmeans/kmeans_functions.py** : 86%
57 statements | 49 run | 8 missing | 0 excluded

Figure 5: Copertura C0 dell'algoritmo di clustering k-means

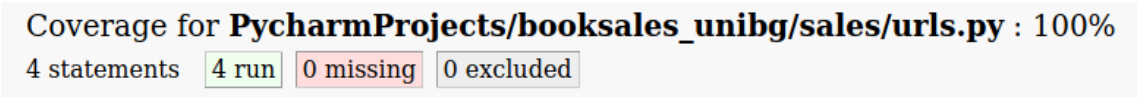

Figure 6: Copertura C0 degli URL della webapp