

DiceIT

Link [GitHub](#) del progetto

Dicet è un sito web gestionale per la vendita di acquisto e dadi da gioco per giochi di ruolo da tavolo, pensato per esporre set realizzati al dettaglio da artigiani e non venduti in massa. Il sito offre la visione dei prodotti in vendita libero per qualunque utente generico, mentre per poter effettivamente acquistare e registrare i set di dadi è necessario registrarsi e accedere al proprio account.

E' poi possibile diventare un artigiano, e avere la possibilità di aggiungere al sito i propri set di dadi, caricando un'immagine rappresentativa, metterli in vendita, modificarne le caratteristiche in seguito e rimuoverne la disponibilità.

Oltre al normale funzionamento di uno store online DiceIT aggiunge la funzionalità delle orde (hoards), una sezione del sito dove gli utenti possono esporre i propri acquisti e visualizzare le collezioni delle altre persone.

Queste collezioni ricevono un punteggio in base al numero di set posseduti, che colloca gli utenti in una classifica globale del sito, con assegnazione di rank per fasce; questa funzione ha lo scopo di incentivare l'aspetto collezionistico dell'acquisto dei dati e spingere l'utente a fare ulteriori acquisti sulla piattaforma.

Casi d'uso

Tipologie di utente:

- utente non registrato
- utente registrato
- utente artigiano

La prima tipologia di utenti è quella **non registrata**, che può accedere al sito web ed esplorare i vari set messi in vendita agli artigiani.

Questa parte dell'esperienza è comune anche agli **utenti registrati**, che però hanno anche accesso alle pagine dei singoli prodotti, e alla possibilità di effettuare l'acquisto; se ciò avviene, il set viene inserito nell'orda(hoard) dell'utente aggiungendolo alla sua collezione personale.

Nello store è possibile per tutti gli utenti fare ricerche sui set per nome, o colore primario del set, restringendo il campo di ricerca sui prodotti di interesse.

L'**utente registrato** può inoltre accedere alla sezione hoard, dove può scegliere se visualizzare la propria collezione, vedendo gli acquisti effettuati in ordine cronologico, oppure esplorare le orde degli altri utenti.

La sezione dedicata all'esplorazione mostra una lista di utenti, enunciando il numero di dadi posseduti e il suo ranking nella classifica, e la possibilità di visualizzare la collezione dell'utente specifico.

Ogni set di dadi esposto, indipendentemente dalla collezione, offre una scorciatoia alla pagina del prodotto, nel caso un utente si interessi all'oggetto nello specifico e voglia comprarlo a sua volta.

L'utente registrato, se non ha fatto alcun tipo di ricerca nello store dei dadi, vede i set **ordinati secondo un sistema di raccomandazione** basato sui suoi acquisti precedenti, se ne ha eseguiti.

Infine, l'utente registrato ha accesso alla pagina di logout, per uscire dall'account.

L'utente registrato, dalla pagina home può poi decidere di diventare **utente artigiano** tramite menù ed essere aggiunto a quella categoria, ottenendo accesso alla sezione banch di DiceIT, dove può visualizzare i set che ha creato, aggiungerne di nuovi, impostando i dati necessari e caricando un'immagine del se esposto, oppure modificarne uno creato in precedenza.

Un set di dadi in vendita, se gli viene rimossa la disponibilità, non sarà più acquistabile dallo store, ma resterà visualizzabile per mantenere la funzionalità di collezionismo delle ordine attiva.

Dopo aver aggiunto un set si viene reindirizzati nella sezioni di modifica dei propri prodotti, nel caso si voglia ricontrillare i valori inseriti o correggere un errore; mentre se si modifica un prodotto si viene reindirizzati alla pagina d'acquisto del set di dadi in questione.

Un utente artigiano, può inoltre compiere ogni altra azione di un utente registrato comune.

Per accedere alle varie operazioni è presente una barra di navigazione dinamica che espone le varie opzioni all'utente in ogni punto del sito, dunque da ogni sezione è possibile raggiungere ogni altra funzionalità in qualunque momento.

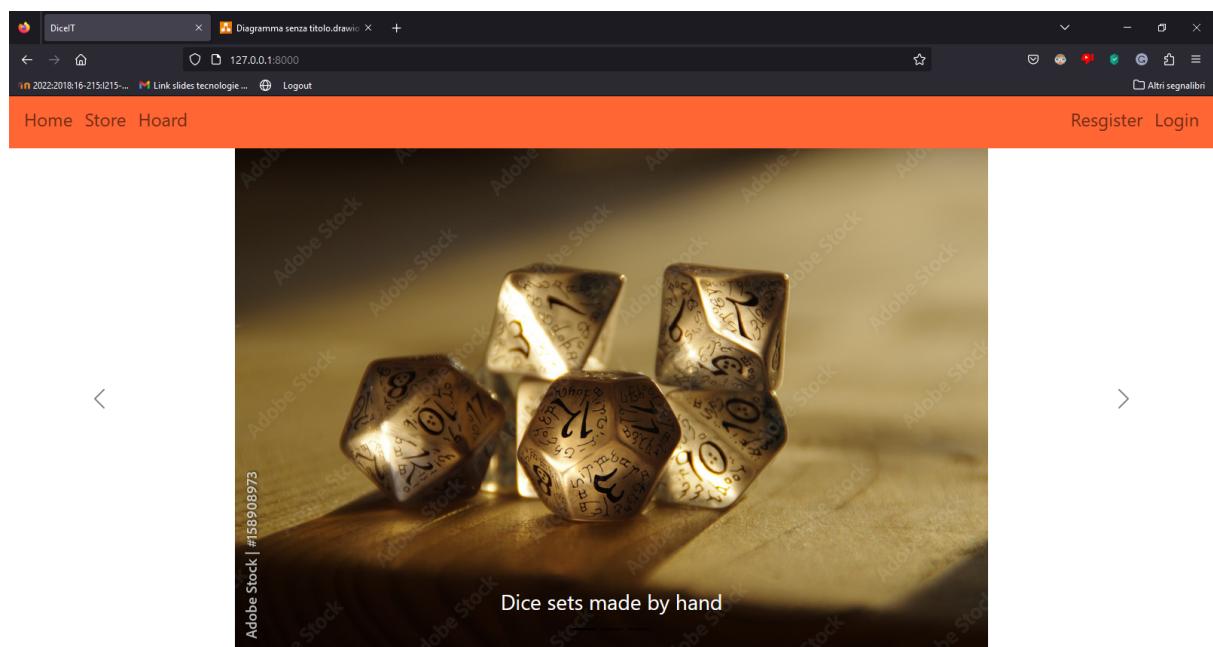
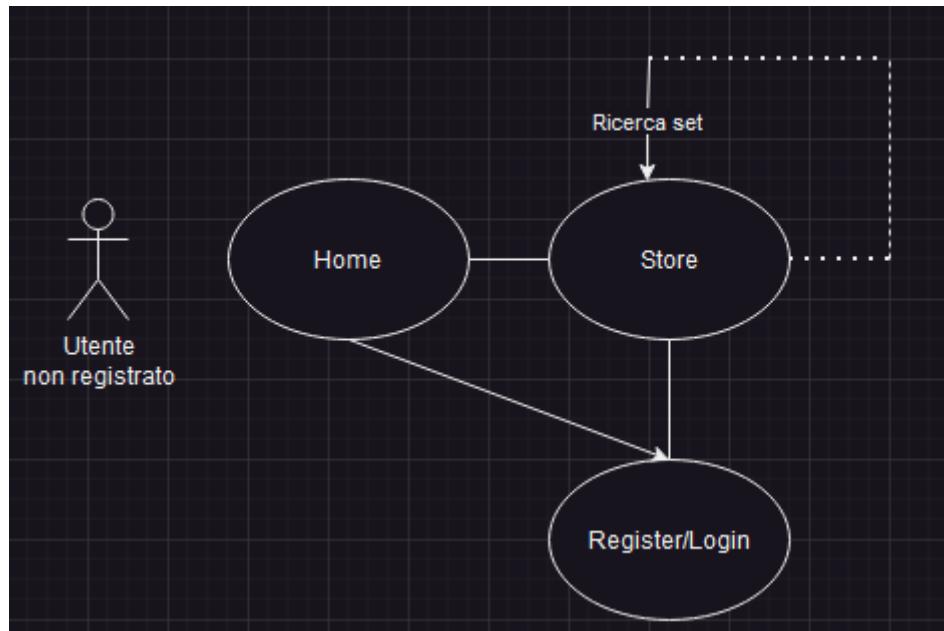
Nello sviluppo della struttura del sito ho cercato di tenere a mente la regola dei 3 click: mantenere la distanza dalla home alla funzionalità desiderata in termini di click del mouse al massimo di 3 unità, inoltre ho voluto mantenere una barra di navigazione globale per non dover mai obbligare l'utente a dover modificare la barra di ricerca per tornare a un punto specifico, o ricominciare il percorso svolto.

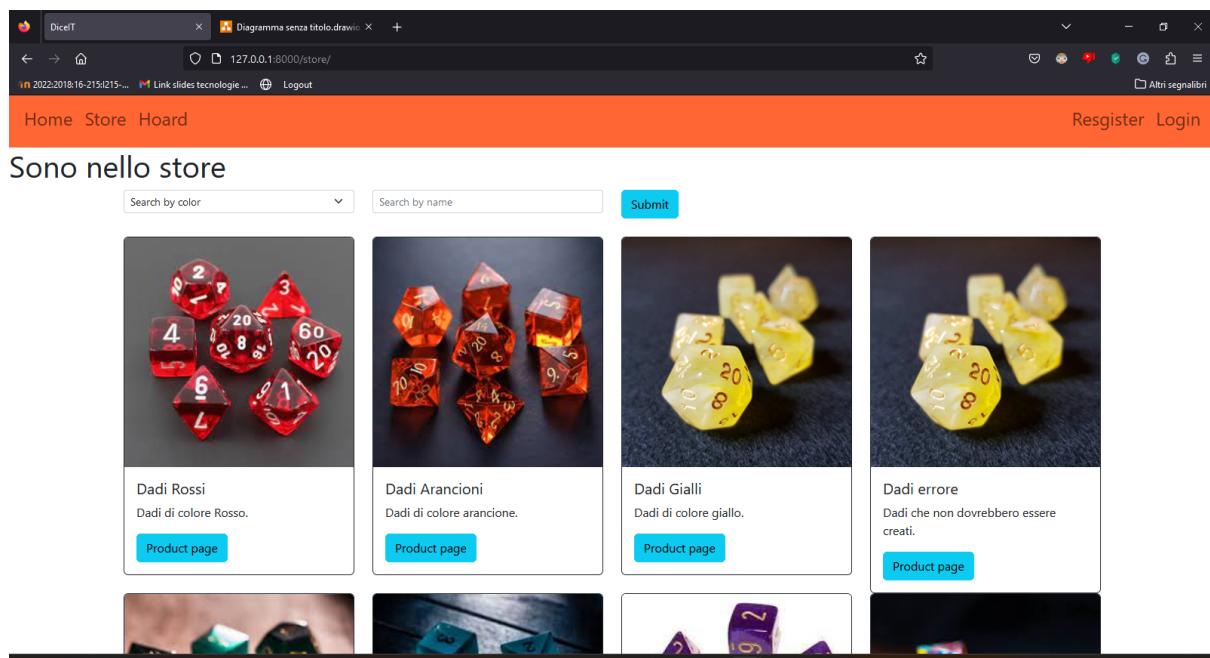
Allo stesso tempo essa nasconde le funzionalità non necessarie, o non disponibili agli utenti che non possiedono i privilegi per utilizzarle, in modo di minimizzare le voci del menu non realmente accessibili, e indirizzando al login in caso serva un account per proseguire in quella funzionalità di DiceIT.

A seguito il diagramma dei casi d'uso del sito con le relative immagini delle pagine di riferimento.

Nota: per questione di leggibilità non ho aggiunto i collegamenti relativi alla barra di navigazione globale del sito, ma per visualizzare il suo lavoro si immagini che ci sia una freccia da ogni sezione del sito che punta al primo blocco che si incontra partendo da home andando verso le macro-aree store, hoard, banch e register/login/logout

Utente non registrato





The screenshot shows a registration form on the DiceIT website. The title bar indicates the page is at `127.0.0.1:8000/store/register/`. The form fields are:

- Username***: A text input field with placeholder text: "Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only."
- Password***: A text input field with placeholder text: "Your password can't be too similar to your other personal information. Your password must contain at least 8 characters. Your password can't be a commonly used password. Your password can't be entirely numeric."
- Password confirmation***: A text input field with placeholder text: "Enter the same password as before, for verification."

At the bottom of the form is a green "Register" button.

DiceIT

Diagramma senza titolo.drawio

2022:08:16-21:51:21... Link slides tecnologie... Logout

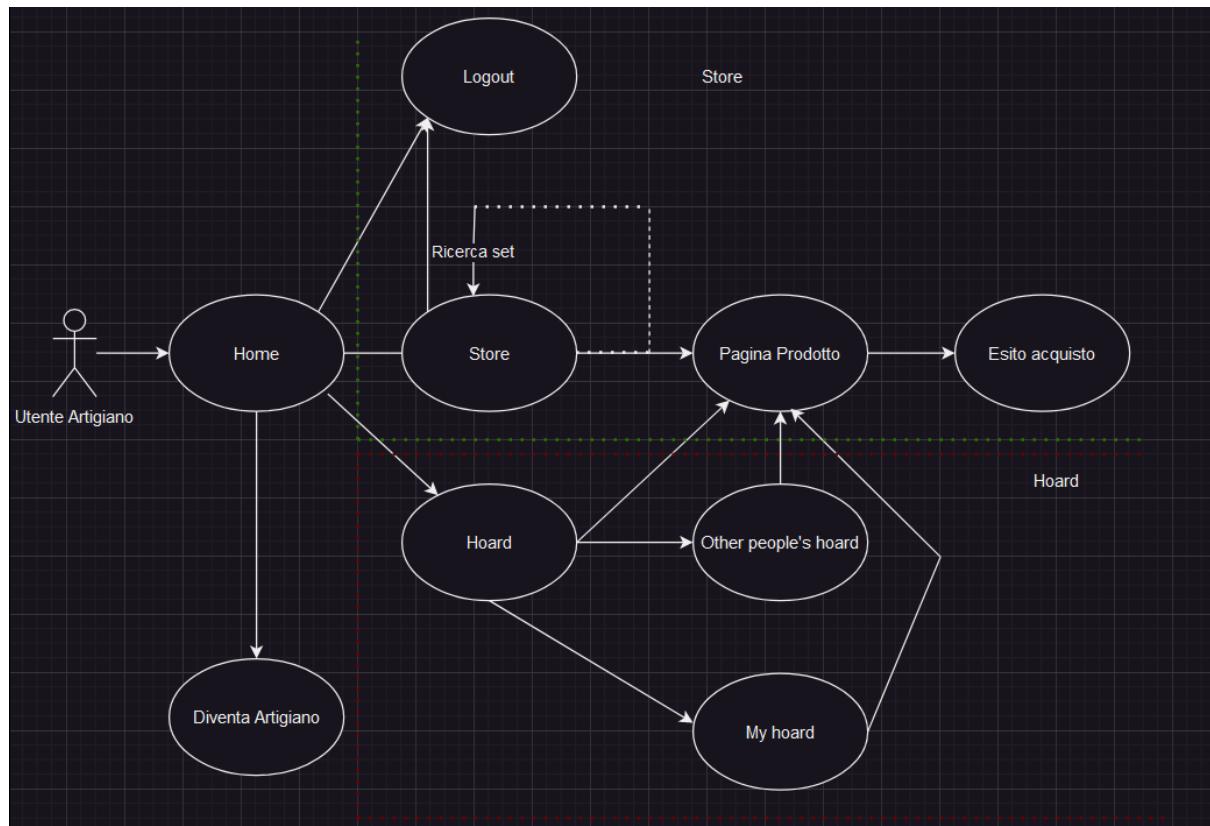
Home Store Hoard Register Login

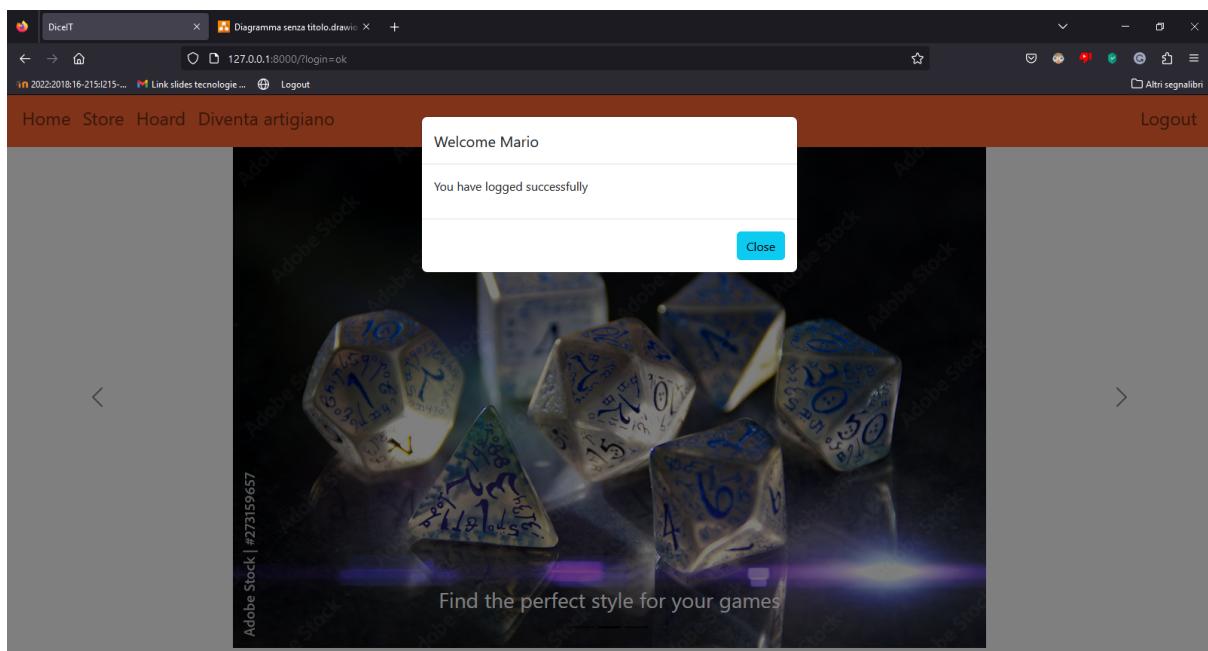
Username*

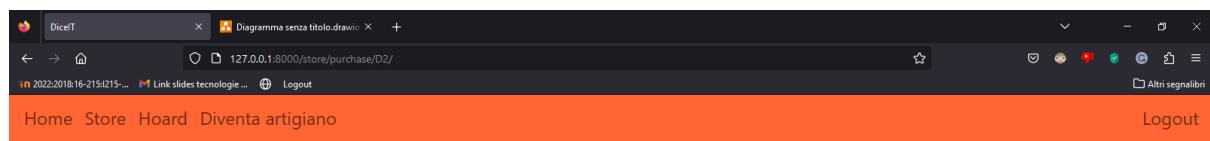
Password*

Esegui login

Utente registrato







Purchase done successfully

It cost you 7.99€



A screenshot of a web browser window titled "DiceIT". The address bar shows the URL "127.0.0.1:8000/hoard/". The page content features two main sections: "Your Hoard" and "Other Hoards". Both sections include a "Go!" button and a small "x" icon. A message at the bottom right says "Hey, you were lucky!!!".

Home Store Hoard Diventa artigiano Logout

Go to see...

Your Hoard

Check your purchases

Go!

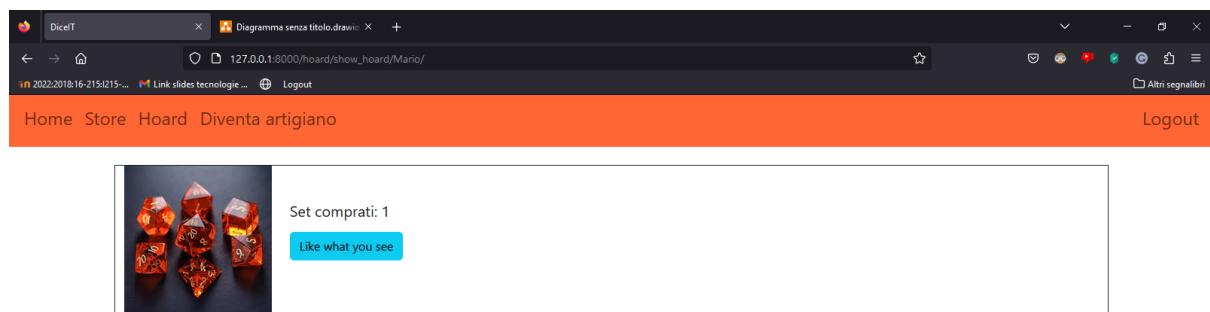
Go to see...

Other Hoards

Take inspiration from other people's collections

Go!

Hey, you were lucky!!!



A screenshot of a web browser window titled "DiceIT". The address bar shows the URL "127.0.0.1:8000/hoard/explore_hoards/". The page content includes a navigation bar with links for Home, Store, Hoard, Diventa artigiano, and Logout. At the top, there are search fields for "Search by name of collector", "Minimum of dices", "Maximum of dices", and a "Submit" button. Below these fields, there are three user profiles listed in cards:

- The Dice Emperor**
The hoard of TheDiceAddicted
Has collected 100 set of dices
Go!
- Dice Dragon**
The hoard of LikeADragon
Has collected 56 set of dices
Go!
- Dice enoyer**
The hoard of admin
Has collected 13 set of dices

DiceIT Diagramma senza titolo.drawio

Logout

Home Store Hoard Diventa artigiano Logout

Set comprati: 10 Like what you see

Set comprati: 10 Like what you see

Set comprati: 10 Like what you see

127.0.0.1:8000/hoard/

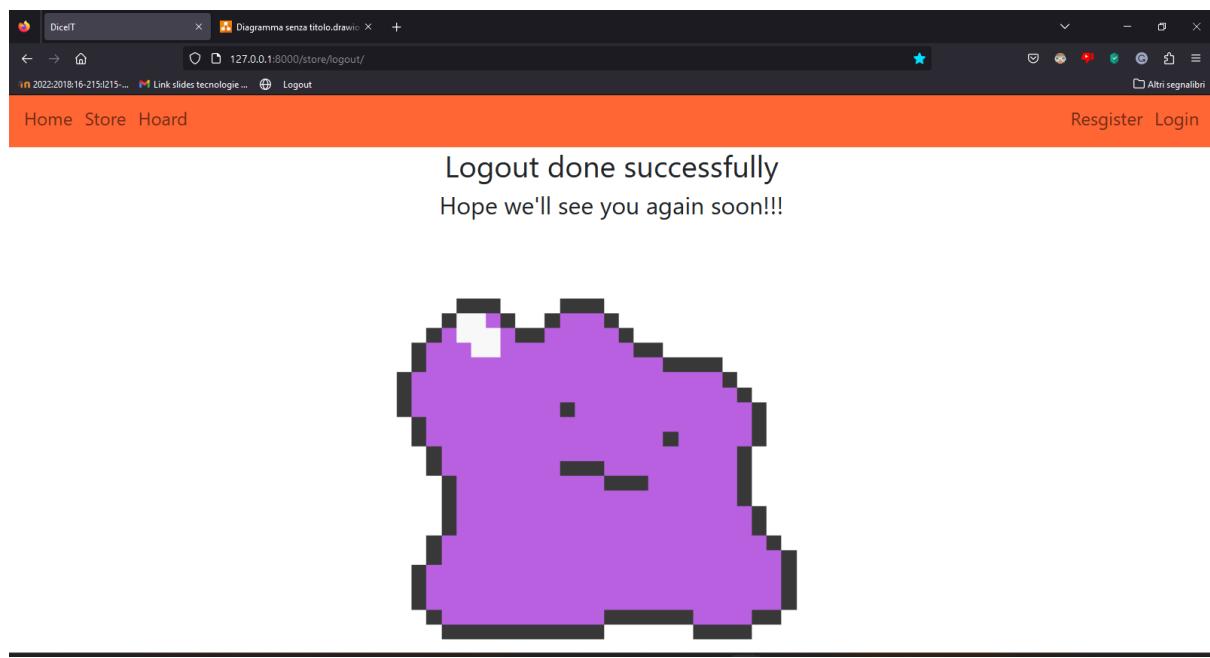
DiceIT Diagramma senza titolo.drawio

Logout

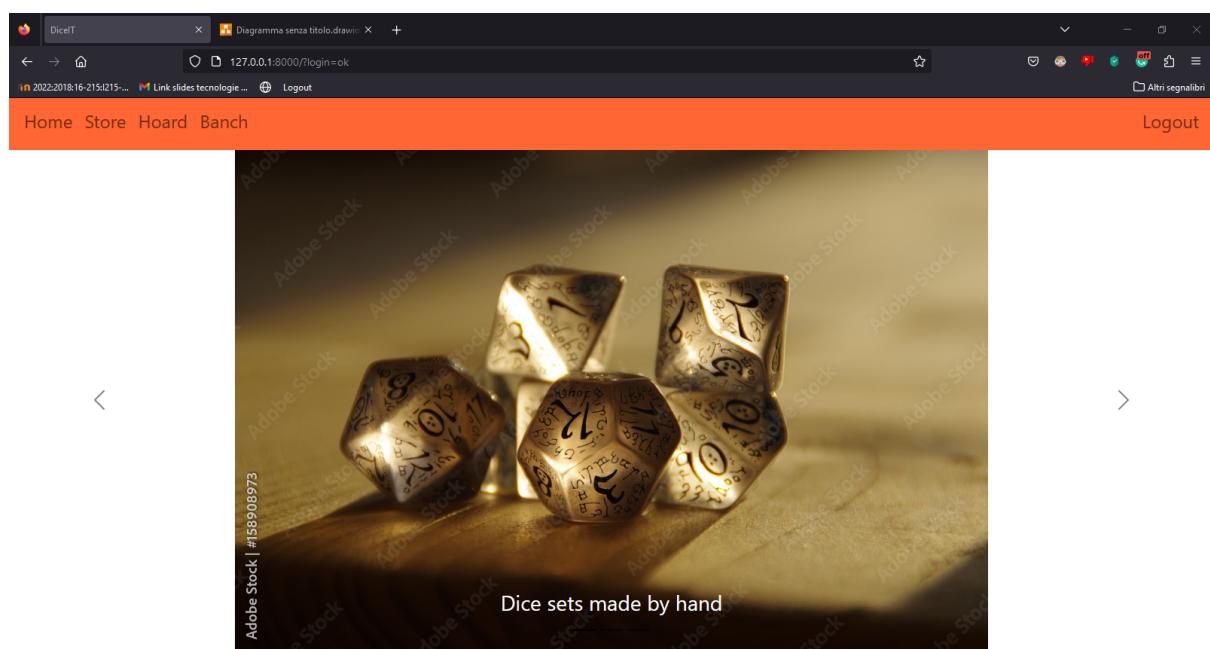
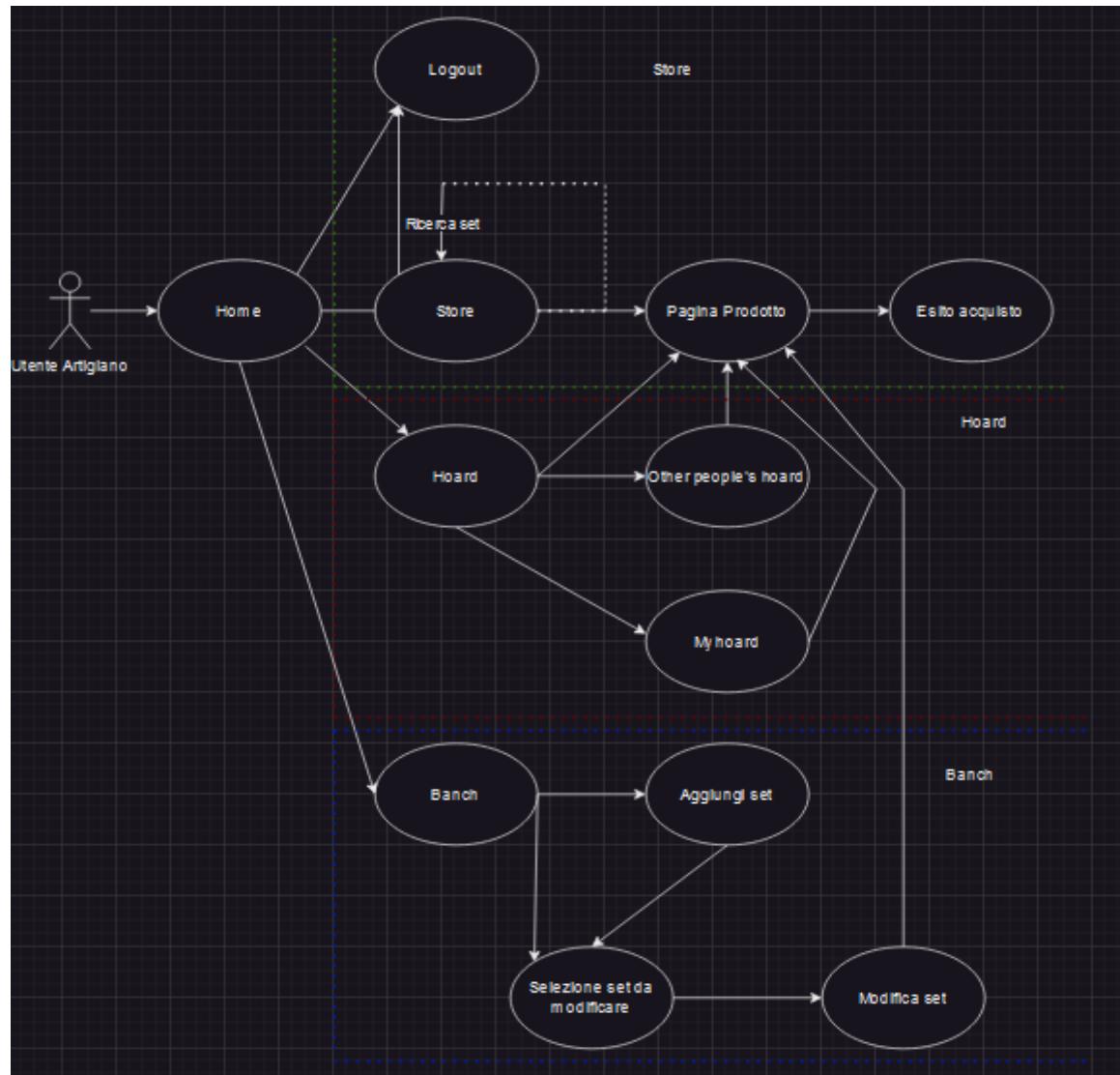
Home Store Hoard Diventa artigiano Logout

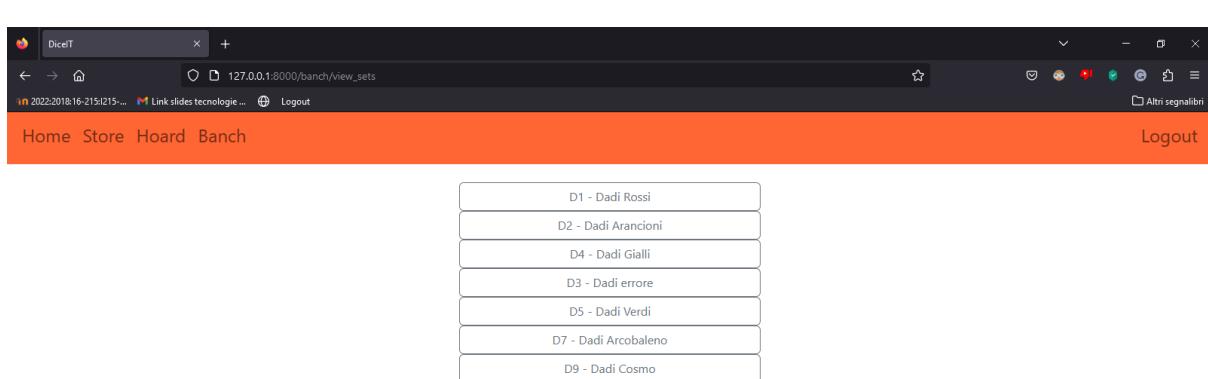
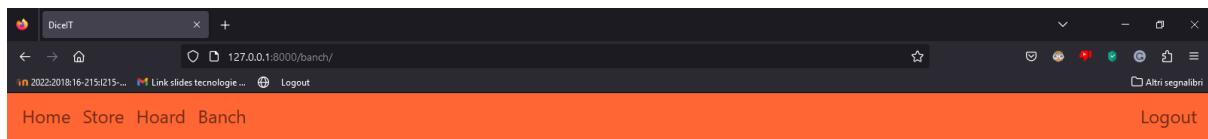
Once an artisan, always an artisan, do you accept it

Accept



Utente Artigiano





Update set ↓

The screenshot shows a browser window for the DiceIT application. The URL is 127.0.0.1:8000/banch/modify_set/D4/update/. The page has an orange header with the navigation links: Home, Store, Hoard, Banch, and Logout. Below the header is a form for updating a dice set:

Name*	<input type="text" value="Dadi Gialli"/>
Number of pieces*	<input type="text" value="7"/> <input type="button" value="..."/>
Primary color*	<input type="text" value="Yellow"/>
Description*	<input type="text" value="Dadi di colore giallo"/>
<input checked="" type="checkbox"/> Available	
Price*	<input type="text" value="7.99"/> <input type="button" value="..."/>

Create set ↓

The screenshot shows a web application interface for creating a new dice set. The page title is "Create set". The form fields include:

- Code* (text input)
- Name* (text input)
- Number of pieces* (text input)
- Primary color* (dropdown menu)
- Description* (text input)
- Seller* (dropdown menu)
- Available (checkbox)
- Price* (text input)
- Image* (file input with browse button and placeholder "Nessun file selezionato.")

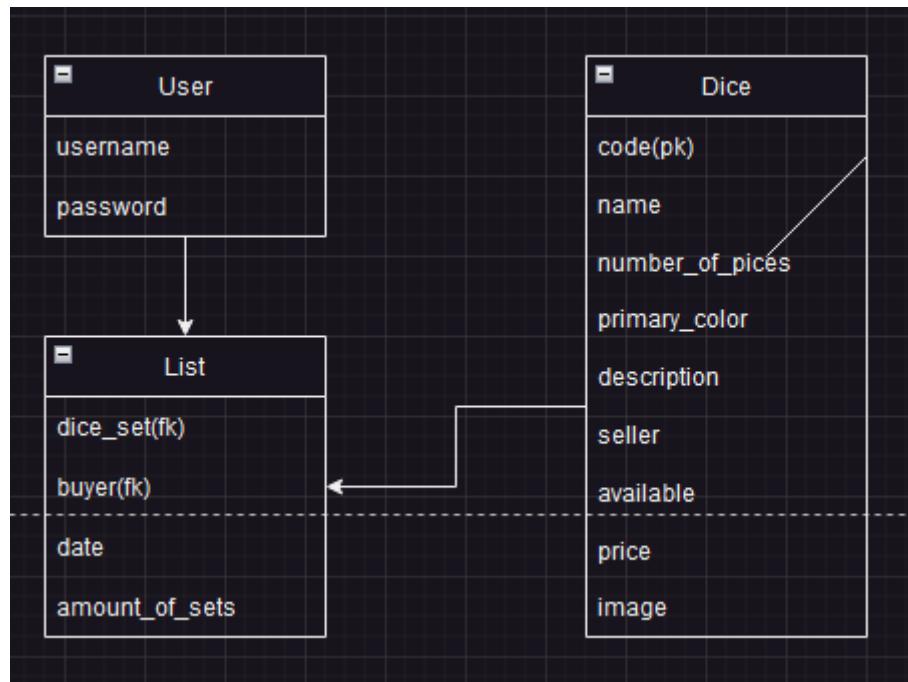
A "Create Set" button is located at the bottom right of the form.

Model del progetto

Ho cercato di mantenere il modello del database il più semplice possibile, sfruttando quello già predisposto da Django per gli utenti: **User**.

Ad esso sono accompagnati le tabelle **Dice**: rappresentante i set di dadi messi in vendita sul sito con vari dati descrittivi e l'immagine di riferimento da caricare quando richiesta; e l'entità **Purchase** che modella l'acquisto da parte dell'utente di uno o più set di dadi di un tipo, mettendo in relazione le entità, nello specifico:

I dadi sono in relazione con gli utenti in due modi: ogni set di dadi ha un utente creati, e indirettamente tramite la tabella Purchase con gli utenti che le acquistano:



User

username: username dell'utente

password: password dell'utente cifrata

Dice

code: codice del dado, chiave primaria dell'elemento

name: nome del set di dadi

number_of_pices: numero di dadi facenti parte del set

primary_color: colore dominante del set, utile per fase di ricerca dei set

description: breve descrizione del set

seller: venditore del set (cardinalità 1 user : n sets)

available: disponibilità alla vendita del set (un booleano)

price: costo di vendita in euro

image: file immagine uploadato dall'utente che mostra il set

nota: ho scelto di non implementare ne nel modello ne nel progetto una tabella per memorizzare gli acquisti poichè non sarebbe mai stata utilizzata per alcuna funzionalità, e da un punto di vista della navigazione avrebbe aggiunto uno step in più alla fase di acquisto che sebbene più realistico, non avrebbe portato alcuna vera nuova funzionalità al progetto.

Purchase (n - m)

dice_set: chiave esterna del set di dadi acquistato

buyer: chiave esterna dell'utente che ha eseguito l'acquisto

date: nella quale è stato eseguito l'acquisto

amount_of_sets: numero dei set di quel tipo acquistati, impostato come massimo di 10

Per ulteriori dettagli sull'implementazione si veda il file **store/models.py**

Tecnologie usate

Segue la lista delle tecnologie utilizzate per sviluppare il progetto:

crispy_forms e **crispy_bootstrap4** sono librerie esterne al progetto aggiunte per migliorare l'aspetto e le funzionalità dei form come vista a lezione, in particolare per le ClassBasedView per le interazioni con il model, dove è risultato molto più semplice e diretto utilizzarle rispetto che a realizzare delle FunctionBasedView.

Tuttavia la maggior parte della grafica del progetto è realizzata da Bootstrap 5, importato tramite link direttamente nei templates, col quale ho sperimentato in maniera più reattiva e con maggiore controllo fino a raggiungere un aspetto che mi soddisfacesse.

scikit-surprise è una libreria di python usata per creare, allenare e utilizzare algoritmi di recommendation system, integrazione fondamentale nel progetto che ordina lo store in base a ciò che viene calcolato possa piacere all'utente registrato in base alla sua similarità con altri utenti basata sui suoi acquisti.

Durante la fase di sperimentazione ho anche utilizzato numpy, ma non è rimasto nella versione finale del progetto.

I tre oggetti usati sono Data, Reader e KENWith Means, per ulteriori dettagli guardare la sezione “Organizzazione logica a livello di codice” e i file diceit/rater.py e diceit/rater.txt .

django-braces e **pillow** sono le librerie usate per eseguire l'upload di file multimediali (nel mio caso immagini) associate ai set di dadi, e salvate in una directory correlata in settings.py.

Ho utilizzato questo metodo per la gestione delle immagini a seguito di svariate di difficoltà mentre cercavo di eseguire l'operazione tramite FBV un form “fatto a mano”, che però è risultato molto più difficile di quanto avevo previsto e che in seguito mi ha spinto a cambiare metodo.

Tecnologie di ricerca usate e fonti principali:

- Documentazione Python
- Documentazione Bootstrap 5
- W3School per python, html, css, javascript e bootstrap
- Chat GPT
- Stack Overflow
- Real Python

Organizzazione logica dell'applicazione a livello di codice

Divisione in applicazioni

Inizialmente pensavo di poter mantenere l'intero processo in un' unica applicazione dato che ritenevo le funzioni da implementare semplici rispetto ad altri progetti svolti nel corso di Laurea, ma la complessità del progetto è cresciuta rapidamente e ho capito presto che era un approccio quanto meno poco pratico, così l'ho abbandonato.

Ho diviso il progetto per macroaree di funzionalità, cercando di raggruppare le funzioni logicamente correlate.

diceit: l'applicazione iniziale, inizialmente conteneva solo la home, poi ho raggruppato al suo interno anche la gestione del recommendation system per svilupparlo in un ambiente “meno affollato rispetto allo store” e controllarlo meglio.

In più ho messo nell'app anche l'aggiunta di un utente al gruppo degli artigiani, per separarla dalla zona del sito alla quale l'utilizzatore non aveva ancora accesso anche se la funzione è logicamente correlata, per questione di ordine.

store: l'applicazione più sostanziale che contiene lo store in sé, le zone di acquisto e l'area di registrazione, login e logout dell'utente; che si trovano qui perchè è la zona di interesse principale del sito e la quale l'utente registrato esegue la maggior parte delle attività.

Nell'app sono svolto l'acquisto, e quindi la creazione di nuove entry nel database, assieme alla registrazione di nuovi utenti, unica CBV; il resto dell'app usa FBV poiché vengono svolte operazioni solo di visita, al database con processing, che quindi ho voluto sviluppare con maggiore controllo sul processo.

Questa è anche la parte che usa il recommendation system e che ho scelto di integrare con delle FBV poiché avevo svolto più esercizio con esse e mi sentivo più sicuro.

hoard: quest'app implementa la parte sociale nel sito, nella quale sono raccolte le informazioni derivanti dagli acquisti degli utenti ed esposte sotto forma di ordine (collezioni). Ho deciso di utilizzare un template per esporre l'orda, modificando un parametro col quale si raggiungeva la view per cambiarne il comportamento; questa è stata la fase del progetto dove ho applicato di più la filosofia DRY.

Inoltre è disponibile una classifica che assegna un rank agli utenti in base ai loro acquisti, e che affibbia al primo un titolo personale, unico nel sito.

banch: app chiamata così in riferimento al tavolo da lavoro dell'artigiano, è l'app del progetto che mi ha richiesto più tempo poiché ho dovuto gestire l'upload delle immagini, molto più complicato di quanto mi aspettassi.

Qui le view sono accessibili solo da utenti artigiani, dove possono modificare o aggiungere nuovi file, qui ho lasciato il più possibile gestire l'interazione col mode e con i media caricati al "Signor Django", limitando ad esporre il form della CBV e a renderlo crispy per l'esposizione.

Ho scelto di non permettere all'utente di modificare l'immagine caricata la prima volta per evitare di gestire un sovraffollamento di immagine poiché è risultato complicato cercare di gestire le immagini caricate.

Alla fine ho scelto di evitare la funzionalità per evitare di creare file inutili nel progetto che sarebbero dovuti essere risolti a mano.

Nota: Avrei voluto correggere il cambio dell'immagine ma una fase di testing problematica e altri progetti da realizzare assieme al carico di studio non mi hanno lasciato il tempo per dedicarmi al problema come avrei voluto.

Scelte implementative

Da static a media

Una delle decisioni principali del progetto è stata dovuta all'upload delle immagini da parte dell'utente, che ha rappresentato una difficoltà imprevista.

Inizialmente utilizzavo la directory **static/** per la visualizzazione delle immagini dei dati, e speravo di poter caricare l'immagine dell'utente, salvarle nella medesima cartella e usare un sistema di risoluzione dei nomi dinamico per usarla nel sito.

Tuttavia non riuscivo ad accedere al file caricato alla view, solamente al nome, e quindi non avanzavo nel progetto; così mi sono confrontato con altri studenti e sui loro metodi per gestire la funzionalità.

Dopo il confronto ho capito che sarebbe stato molto più semplice smettere di provare a gestire l'upload di un file e lasciare che fosse Django a farlo, così ho modificato il Model dei dati aggiungendo un ImageField.

In questo modo il modello si occupa di gestire la posizione del file, e il suo recupero quando richiesto e la ClassBasedView per la creazione di un nuovo oggetto, specificando il modello, ed esso esegue tutte le operazioni necessarie, mascherando i vari passaggi.

Con questa scelta però ho dovuto rinunciare alla generazione automatica del codice della nuova entry nel database: questo perché giango ottimizza la memoria generando una sola volta l'oggetto CBV per l'inserimento del set nel database e poi utilizzandolo, e quindi, richiamare una funzione per generare un codice nella classe è un'operazione che viene eseguita una sola volta.

Ne consegue che ricaricando la pagina più volte si ottiene sempre lo stesso codice, e quindi un oggetto non inseribile nel database.

Ho provato ad aggirare il problema in molti modi, anche con Javascript, ma non riuscivo ad accedere al campo del form generato dal template dopo il caricamento, né a renderlo dinamico prima.

Alla fine ho rinunciato lasciando il campo da riempire all'utente, con i controlli automatici a garantire l'integrità: not null, unico ...

Recommendation System

Ho scelto di usare un recommendation system **user-based** poichè nel mondo dell'artigianato esistono migliaia di varianti di oggetti apparentemente uguali come i dati: ci sono versioni diverse di materiale, colore, forma, decorazioni interne ed esterne, tipi di set, tipi del dato specifico ...

In questo ambiente variegato ho pensato che sia più sensato cercare di prevedere le preferenze del cliente cercando di capire che tipo di collezionista è; basando i suggerimenti che hanno fatto acquisti con pattern simili ai suoi.

Il modulo python per realizzare il sistema di raccomandazioni che ho scelto è **scikit-surprise**, che fornisce vari oggetti e funzionalità legate a questo aspetto, personalizzabili e semplici da integrare al proprio codice, aspetto importante alla base della mia decisione.

Ho usato gli oggetti **Dataset**, **Reader**, **KNNWithMeans**:

Reader per ottenere i dati per allenare il predittore tramite un file che ho creato primita tramite una view accessibile solo da un utente amministratore, poi tramite una funzione richiamata all'avvio del sito, e una volta ogni intervallo di tempo prefissato tramite variabile globale in settings.py.

Reader fornisce gli strumenti per leggere i dati da una sorgente specifica (file, data frames, dizionari...) separando logicamente il tipo di lettura da effettuare dalla raccolta dei dati.

Il file file **ratings.txt** contiene i dati in formato <utente item rating>, il rating è ottenuto convertendo il numero di acquisti di un determinato set in una scala da 1 a 10.

Dataset è l'oggetto che trasforma i dati letti in un frame di dati ottimizzato per la creazione dell'oggetto che calcolerà il rating stimato quando invocato; è stato creato specificando il file da

cui raccogliere le informazioni e l'oggetto Reader che si è occupato degli aspetti pratici della lettura, inizializzato in precedenza.

KNNWithMeans: crea l'oggetto predittore che viene generato passandogli un dizionario di parametri nel quale si specifica il tipo di misura di similarità scelta, nel mio caso "cosine" similarity, e il tipo di recommendation da calcolare, quindi "user_based" che esegue le predizioni basandosi sulla similarità da utenti.

L'oggetto restituito dalla chiamata viene poi allenato sul training set ottenuto dai dati generati dal dataset, producendo l'oggetto finale.

Nel sito il predittore è allenato ogni volta che viene invocata la view di visualizzazione dello store, per avere i dati più aggiornati possibile: questo non è un approccio che scala con le dimensioni del sito, in quanto l'aumento dei dati disponibili renderebbe l'operazione più lenta.

Ci sono approcci più indicati come circoscrivere i dati raccolti ai più recenti, oppure aggiornare il predittore con meno frequenza, parallelamente al suo utilizzo, ma visto che il progetto non è destinato a crescere ho scelto un approccio ispirato alla realtà, ma più semplice e facile da integrare.

Inoltre il file su cui sono salvati i dati degli acquisti è rigenerato ciclicamente all'avvio del sito, ad intervalli di tempo regolari, o quando un nuovo set di dati è creato e aggiunto al database, in modo da fornire sin da subito una predizione accurata sull'oggetto.

Test

I due test che ho scelto di scrivere riguardano il recommendation system per la parte funzionale, poiché per me rappresenta la parte più importante del progetto e che mi ha spinto maggiormente fuori dalla mia area di competenza con le ricerche.

Invece il testing tramite client ho deciso di controllare le aree del sito con permessi, assicurandomi che l'utente venisse redirezionato qualora tentasse di aggirare la barra di navigazione.

I test si trovano nel file store/tests.py, e sono suddivisi in 4 funzioni; le prime tre sono relative al testing funzionale e fanno riferimento ad un oggetto predittore inizializzato con la classe, lo stesso che il sito userebbe normalmente, inizializzato dalla stessa funzione in diceit/rater.py .

Eseguendo un black box testing prima provo a passare all'oggetto dati corretti, e poi dati errati, ma nel formato corretto: <stringa : user> <stringa : item> .

L'oggetto deve restituire in entrambi i casi un valore numerico affinché il programma continui l'esecuzione, ma con i dati errati restituisce anche informazioni in un dizionario che indicano il fallimento della predizione.

Il terzo test invece puntava a "rompere" il predittore, passandogli oggetti di vario tipo (date, dizionari, tipi), ma essendo scikit-surprise, un modulo già diffuso stabilmente da molto tempo è in grado di gestire in coerenze e l'oggetto predittore ha funzionato correttamente in tutti i casi.

Infine nel quarto test ho usato l'oggetto client per tentare l'accesso tramite request a varie pagine del sito, aspettandomi il messaggio 302, e il redirect alla pagina di login, in quanto la mancanza dei permessi è risolvibile dalla creazione di un account, o dal suo cambio.

Problemi riscontrati

Uno dei problemi che non sono riuscito a risolvere è stato creare campi precompilati e/o non editabili nei form delle Class Based View per la creazione di nuovi set di dati, usando le classi preesistenti di django; in particolare per il codice del nuovo set di dati e per il venditore.

Cercare di alterare il comportamento del form non ha dato risultati poiché o l'oggetto dava errori all'avvio del server oppure il form generato al caricamento della pagina non era accessibile dagli script che eseguivano al momento del caricamento.

Alla fine, visto che la data di consegna era vicina, e la view funzionava correttamente ho deciso di lasciare l'errore e riportare il resoconto dei tentativi di correzione in questa sezione.

L'altra fase del progetto che ha dato più problemi è stato l'uso dei file multimediali poiché ho dovuto cambiare logica di gestione delle immagini, che prima trattavo come risorse statiche e che recuperavo tramite una risoluzione dinamica dei nomi.

Usando invece un campo del modello pensato apposta e cambiando alcuni settings ho potuto evitare di preoccuparmi del recupero delle immagini, lasciando che la logica di Django se ne occupasse.

Ma prima di usare questo approccio ho perseverato con la mia idea iniziale per molte ore di lavoro poiché non volevo cambiare il modello temendo che questo avrebbe causato problemi a catena in tutto il progetto.

In realtà le mie paure erano dovute all'inesperienza con django, e quando mi sono deciso a modificare approccio ho dovuto solo aggiungere una riga nel modello Dice, una per ogni view di riferimento e un riferimento nei Templates.

Questa fase del progetto mi ha mostrato di prima mano quanto realmente Django sia pensato per essere semplice da usare e modificare, e che le conoscenze degli strumenti che utilizzi per sviluppare un progetto sono fondamentali per poterli usare appieno.

Anche il testing non è stato semplice come speravo, poiché l'uso di raccomandazioni creati a partire da modelli non inizializzati durante i test dava problemi col set-up, che ho risolto tagliando fuori dall'esecuzione quella parte, lasciando che il rating avvenisse sulla base di dati creati in precedenza.

Nota finale: ho lasciato nella cartella static le immagini allo scopo di usarle per creare nuovi set di dati durante le prove del sito e l'esame nel caso mi venga chiesto di mostrare quella funzione.