

Progetto tecnologie web

Gabriel Panini 152348

Luglio 2023

Indice

1	Traccia	1
2	Tecnologie usate	3
3	UML use case	4
4	Scelte implementative	5
4.1	utenti_custom	6
4.2	gestione_utenti	7
4.3	gestione_medici	10
4.4	gestione_assistenza	12
4.5	autorizzazioni e gruppi	13
4.6	email	14
4.7	Pagina admin	14
5	Modelli	15
6	Test	16
7	Schermate aggiuntive	17
8	Problemi riscontrati	18

1 Traccia

Sistema di prenotazione di esami e visite mediche

Applicazione web che gestisce un sistema di prenotazione di esami e visite mediche. L'applicazione è pensata per essere usata da quattro tipi di utenti:

- gli utenti anonimi possono solo cercare esami.

- gli utenti registrati oltre a poter effettuare prenotazioni, hanno una pagina dedicata in cui possono vedere tutte le prenotazioni fatte in passato, cancellare quelle che ancora non state effettuate e scaricare un pdf come promemoria della prenotazione. A seguito della corretta prenotazione, deve essere mandata una mail all'utente di conferma.
- I medici che offrono prestazioni, devono essere registrati al sito prima di poter inserire le visite che possono eseguire. Per ogni visita deve essere indicato il giorno e la tipologia tra una serie di tipologie prestabilite. I medici devono avere anche una pagina dedicata in cui oltre a poter modificare le proprie informazioni, possono aggiungere una loro descrizione. Inoltre devono avere una pagina personalizzata in cui possono vedere gli esami che hanno caricato nel sistema e devono poterli filtrare in base ad una data oppure in base al loro stato. In questa pagina i medici possono cambiare i dettagli degli esami o eliminarli. In entrambi i casi deve essere mandata una mail all'utente che si era prenotato per informarlo del cambiamento/eliminazione.
- L'assistenza clienti ha il compito di registrare i nuovi medici e ha l'accesso ad una pagina specifica per chattare con i clienti i cui dettagli sono specificati in seguito.

Il sistema deve consentire la ricerca di un certo esame in base alla disponibilità in un certo intervallo temporale (la classica scelta "da" "a" presenti in molti siti) e/o se offerte da un certo medico: gli esami trovati in seguito alla ricerca devono essere mostrati in ordine crescente di giorno e a parità di giorno, in ordine crescente di orario (esempio: prima una visita il 10/06/2023 alle 09:00, poi una il 10/06/2023 alle 10:00, infine una il 12/06/2023 alle 08:00).

Il sistema deve gestire la disponibilità delle visite, oscurando agli utenti quelle già prenotate e rendendole di nuovo visibili in caso di eliminazione della prenotazione. Nel caso la prenotazione venga cancellata nello stesso giorno in cui si sarebbe dovuto svolgere l'esame, allora non verrà resa di nuovo visibile agli utenti. Quando un utente effettua una prenotazione, deve essere possibile cliccare sul nome del medico che offre la prestazione e questa azione porterà ad una pagina in cui sono presenti le sue informazioni e una serie di recensioni. Le recensioni possono essere inserite solo dagli utenti registrati.

La registrazione e autenticazione deve avvenire sulla base di una email e password e non con uno username, che non deve essere presente.

Per agevolare l'assistenza clienti, la pagina riservata agli admin non deve limitarsi ad elencare le istanze dei vari modelli, ma deve implementare per ogni modello la ricerca o filtraggio sulla base di attributi significativi.

Infine deve essere presente una pagina di assistenza in cui l'utente possa chattare in tempo reale con un membro dell'assistenza clienti. I membri dell'assistenza clienti devono avere una pagina personalizzata in cui è possibile vedere tutte le persone in attesa di assistenza e schiacciando su un bottone potranno unirsi alla stanza in cui si trova l'utente. Una volta che un membro dell'assisten-

za clienti si unisce alla stanza, questa deve scomparire dalla pagina che mostra gli utenti in attesa di assistenza.

2 Tecnologie usate

In questa sezione indico le varie tecnologie e librerie usate per realizzare il sito:

- **pipenv**: ambiente virtuale utilizzato per sviluppare il progetto;
- **django**: è il framework usato per il backend;
- **bootstrap 5.3**: è la libreria usata per il front-end;
- **django-bootstrap-datepicker-plus**: è una libreria che ho usato per mostrare un calendario quando chiedo ad un utente di inserire una data perché lo trovo più intuitivo rispetto a scriverla a mano;
- **xhtml2pdf**: è una libreria che ho usato per creare i pdf a partire da un file html;
- **django-channels**: è una libreria che ho usato per implementare i websocket necessari alla chat in tempo reale;
- **django-braces**: è una libreria che aggiunge dei mixin per aiutare a controllare nelle class-based view che un utente abbia le credenziali necessarie per accedere ad una certa funzionalità;
- **django-dirtyfields**: è una libreria usata per tenere traccia dei campi dirty dei modelli, ovvero campi che sono stati cambiati ma non ancora salvati su file. La libreria tiene anche traccia del valore precedente. Utilizzata per aiutare l'implementazione dell'invio mail quando veniva modificato un certo campo;
- **django-crispy-forms**: libreria usata per migliorare l'aspetto dei form;
- **tz-local**: è una libreria che fornisce informazioni sui fusi orari e che ho utilizzato per ottenere il fuso orario italiano per passarlo alla funzione *now* del modulo *datetime* per aggiungere il fuso orario alla data. Questo è stato necessario per fare i confronti con le date salvate nei modelli di Django perché contengono il fuso orario indicato nel file *settings.py* e non permettono di fare confronti con date che non li hanno;
- **pillow**: libreria usata per far funzionare l'upload delle immagini;
- **daphne**: server ASGI che ho utilizzato perché il development server non funzionava correttamente con i websocket;
- **sqlite**: database utilizzato di default da Django. Non l'ho cambiato perché l'applicazione non deve scalare.

3 UML use case



Figura 1: diagramma UML use case parte 1

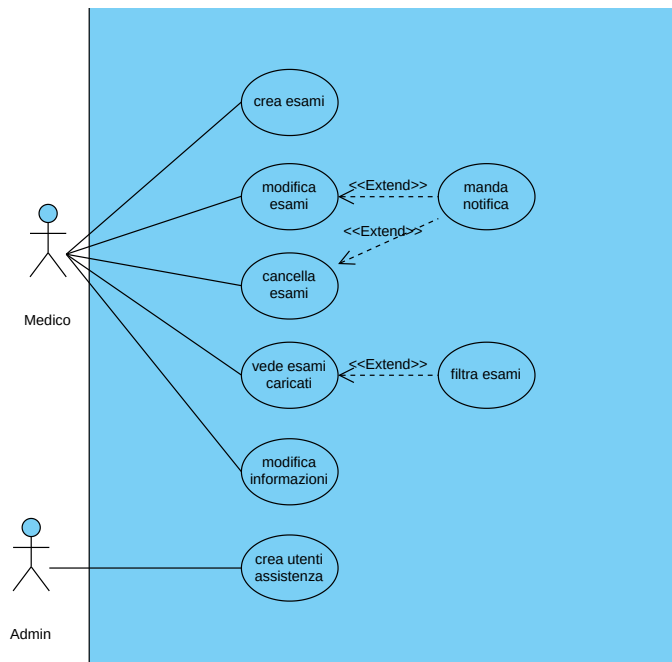


Figura 2: diagramma UML use case parte 2

4 Scelte implementative

La prima scelta progettuale è stata la divisione in app del progetto. In particolare ho deciso di dividerla in 4 app diverse per separare le varie funzionalità relative a diversi utenti nell'ottica che in un sito reale, tenere tutto in una singola app avrebbe reso lo sviluppo disordinato e difficile al crescere delle richieste per nuove funzionalità.

Inoltre in questo modo è più facile testare e debuggare eventuali errori.

Infine in caso di errori introdotti in un'app, le altre app possono continuare a funzionare. È una scelta che facilita anche me nello sviluppo perché mi permette uno sviluppo incrementale chiaramente scandito (nel senso che prima posso iniziare a sviluppare le funzionalità per i medici, poi quelle di ricerca per gli utenti e così via).

Posso dire a sviluppo concluso che questa divisione si è rivelata azzeccata e non ci sono stati grossi intoppi o effetti collaterali mentre implementavo le funzionalità di una certa app.

Devo riconoscere però che le app non sono completamente indipendenti, ad eccezione dell'app per gli utenti custom, perché a volte utilizzano modelli o pagine definite in altre app come per esempio il modello *Esame* definito nell'app *gestione_medici* ma utilizzato anche dall'app *gestione_utenti*.

4.1 utenti_custom

La prima app sviluppata è stata quella per gli utenti personalizzati senza username. Una implementazione più semplice e veloce della mia sarebbe stata quella di usare il modello di default per gli utenti fornito da Django, ignorare il campo username riempiendolo con caratteri casuali (in quanto è la chiave primaria e va riempita) e creare solo il sistema di autenticazione con l'email e password.

Ho deciso invece di implementare un mio modello personalizzato in quanto ritengo che fosse uno spreco avere un campo nel modello che non viene utilizzato mai, inoltre le email non sarebbero state uniche e infine ho seguito le indicazioni della documentazione ufficiale di Django che consiglia all'inizio di un nuovo progetto di creare un modello custom: <https://docs.djangoproject.com/en/4.0/topics/auth/customizing/#auth-custom-user>. Questo mi ha permesso anche di aggiungere un campo per le immagini al modello senza dover creare un modello proxy con conseguente overhead.

Per l'autenticazione è stato sufficiente assegnare alla variabile `USERNAME_FIELD` il campo `email` nella definizione del mio modello e assegnare alla variabile `AUTH_USER_MODEL` presente nei settings il nuovo modello per gli utenti. In questo modo ho potuto usare il backend di autenticazione di default senza problemi.

```
class UtenteCustom(AbstractUser):
    """ Definisce un utente personalizzato senza il campo username e con un campo
    | per l'immagine profilo """

    username = None
    email = models.EmailField(_('email address'), unique=True)
    foto_profilo = models.ImageField(upload_to='immagini_utenti/%Y/%m/%d/', blank=True, null=True)

    # campo che specifica l'identificatore unico da usare per il login
    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['first_name', 'last_name',]

    objects = UserManager()

    def __str__(self):
        """ Ritorna stringa con nome e cognome per rappresentare l'utente """
        return f"{self.first_name} {self.last_name}"
```

Figura 3: modello utente

L'unico lato negativo di questo approccio è che ho dovuto implementare anche uno User manager personalizzato per togliere i riferimenti allo username presente in quello di default perché altrimenti operazioni come la creazione di un superutente non sarebbero funzionate. Tuttavia è stata un'operazione abbastanza veloce visto che ho potuto seguire la falsariga di quello implementato di default.

Il resto dell'app contiene le varie implementazioni per le operazioni standard come registrazione, login, logout e modifica informazioni dove in generale ho creato un crispy form che ho usato in una class-based view e ho spesso usato

il `SuccessMessageMixin` per impostare i messaggi di successo per informare gli utenti.

Due aspetti particolari di cui vale la pena parlare sono l'implementazione dell'invio mail al momento della registrazione e il link per il cambio password contenuto nel form per la modifica delle informazioni associate ad un utente.

Per l'invio della mail l'implementazione è stata abbastanza semplice e ho scelto di fare un override del metodo `save` del form per inviare una mail all'indirizzo inserito dall'utente dopo averlo salvato nel database usando la funzione `send_mail` fornita da Django.

Il problema del link del cambio password invece si è presentato perché per la creazione del mio form ho ereditato da `UserCreationForm` ma quest'ultimo contiene un link verso una pagina per cambiare la password che andava modificato per puntare verso la mia view invece che ad un link di default. Per capire dove veniva definito questo link sono andato quindi a controllare il codice sorgente del form e ho così potuto definire un nuovo metodo `__init__` (perché lì veniva definito) dove il link fosse corretto.

4.2 gestione_utenti

Questa app contiene tutte le operazioni che possono effettuare gli utenti non registrati e registrati. È sviluppando le view per questa app e l'app `gestione_medici` che ho capito bene quando usare una function view e una class view in quanto qui per esempio ho implementato la view per far vedere ad un utente la situazione degli esami con una function view ma l'implementazione del paginatore risulta molto più facile con una `ListView` dove è sufficiente specificare quante elementi per pagina si vogliono con la variabile `paginate_by` invece di dover importare il paginatore, istanziarlo, recuperare il numero di pagina e aggiungerlo alle variabili di contesto. Le view più classiche come quelle per la prenotazione di un esame o la cancellazione sono state implementate facilmente: è sufficiente recuperare l'esame giusto e cambiargli lo stato. Più interessanti invece le view per la stampa degli esami e per la loro ricerca.

Per la stampa degli esami ho deciso di utilizzare una libreria chiamata `xhtml2pdf` per convertire un html in pdf. Avevo provato ad usare anche la libreria `reportlab` ma l'ho trovato più complessa e macchinosa da utilizzare per ottenere un pdf con la giusta struttura. Con la libreria che ho scelto è stato sufficiente creare un semplice html con i vari campi dell'esame e poi passare al metodo `CreatePDF` della libreria il render del mio template. Questo metodo di creare pdf è molto comodo anche perché se in futuro si volesse modificare la sua struttura, basterebbe cambiare l'html lasciando invariato il codice della view.

Promemoria per il suo esame

Gentile utente, ecco il promemoria per la sua visita:

Tipologia: Risonanza magnetica

Medico: Luigi Bianchi

Data e orario: Giovedì 27 Luglio 2023 16:33

Paziente che ha prenotato la visita: Mario Rossi

Figura 4: promemoria

La ricerca degli esami invece è stata più difficile e una delle parti meno riuscite in quanto sebbene funzioni, non trovo molto elegante il modo in cui l'ho implementata. Per realizzarla ho diviso la logica della ricerca in due view: nella prima mostro all'utente un form con tutti i vari campi di ricerca e quando il form ritorna indietro con una richiesta *POST*, controllo che sia valido e redirezione la richiesta verso la seconda view con i vari campi estratti dal form. A questo punto recupero tutti gli esami disponibili e per ogni campo inserito, filtro il queryset. Ho deciso di mostrare tutti gli esami disponibili se non vengono immessi parametri in quanto ho pensato che un utente che non ha mai usato il sito potrebbe essere interessato a vedere i possibili esami che offre il sito, i vari medici che li eseguono e in generale i vari orari. La parte che non mi piace riguarda sia il passaggio degli elementi da filtrare, visto che mi sono limitato ad aggiungere all'url per la seconda view, tutti gli elementi come argomenti, ottenendo un url molto lungo. Inoltre all'interno della view c'è una serie di *if* per controllare che i valori non siano nulli prima di eventualmente filtrare il queryset. Questo non aiuta anche una possibile futura espansione con altri elementi per filtrare, in quanto il numero di argomenti dell'url e di *if* potrebbe diventare troppo elevato.

Cerca degli esami

Per vedere tutti quelli disponibili non inserire parametri

Inserisci il nome del medico

Inserisci il cognome del medico

Da quando?

Fino a quando?

Che tipo di visita?

[Cerca](#)

Figura 5: pagina di ricerca

[Home](#) [Cerca esami](#) [Esami prenotati](#) [Assistenza clienti](#) Registrati  Utente anonimo

La tua ricerca ha dato come risultato 4 esami

Esame disponibile per la prenotazione

Esame della tipologia Pediatria

Un esame di [Paolo Ferrari](#) che verrà eseguito in data Sabato 22 Luglio 2023 17:03

Prenota

Esame disponibile per la prenotazione

Esame della tipologia Risonanza magnetica

Un esame di [Luigi Bianchi](#) che verrà eseguito in data Domenica 23 Luglio 2023 15:00

Prenota

Figura 6: pagina di risultati sopra

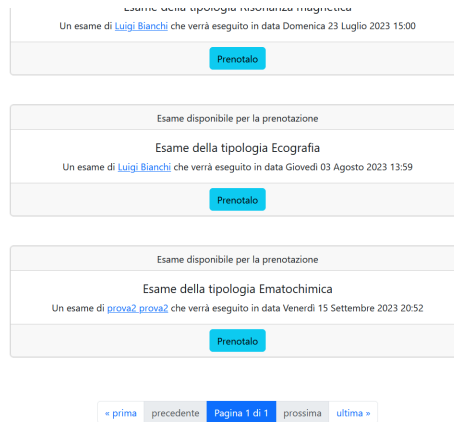


Figura 7: pagina di risultati sotto

4.3 gestione_medici

Quest'app contiene le operazioni che possono svolgere i medici e vari modelli usati anche da altre app come il modello per l'esame. In generale ho usato soprattutto delle class-view in quanto le operazioni dei medici sono abbastanza standard, anche se ho dovuto fare spesso override degli metodo predefiniti per aggiungere condizioni aggiuntive come per esempio un controllo nel metodo *post* della classe che implementa la view per lasciare un commento per impedire ad un utente di commentare più volte uno stesso medico, oppure nella view per la pagina di visualizzazione delle informazioni di un certo medico dove ho dovuto aggiungere alle variabili di contesto anche i suoi commenti.

Una funzionalità particolare di cui voglio parlare è il filtraggio degli esami all'interno della pagina per i medici che gli mostra quali esami hanno caricato. La realizzazione più semplice sarebbe stata avere due pagine separate per la ricerca e per la visualizzazione ma ho pensato che fosse molto macchinoso per i medici doversi spostare tra due pagine diverse ogni volta quindi ho deciso di cercare di realizzare tutto in una singola pagina. Per farlo ho utilizzato un form con tutti i possibili campi e qualche funzione in javascript, che permettono di nascondere e mostrare solo i campi corretti in base a cosa il medico vuole filtrare: se vuole filtrare in base alla data, ci saranno solo i due campi per l'inizio e fine data, se vuole filtrare in base allo stato, ci sarà un campo in cui selezionare il giusto stato e infine un'opzione per rimuovere i filtri. Ad ogni richiesta della pagina quindi, recupero i campi del form e poi in base alla categoria di filtraggio, considero solo i campi associati e infine mostro i risultati all'utente. Ho dovuto però poi risolvere un problema riguardo la paginazione in quanto i parametri vengono passati non come argomenti dell'url ma come query string e quindi con la paginazione classica dove mi limitavo ad associare ai bottoni per muoversi tra le pagine una query string con il numero della pagina desiderato, andavo a perdere i parametri. La soluzione quindi è stata aggiungere all'inizio della

view la creazione di una stringa in formato corretto con gli eventuali parametri presenti nell'url per non perderli quando un utente si muove tra le pagine.

The screenshot shows a green header with the text "Stato attuale dei tuoi esami caricati". Below it, on the left, is a filter section with the label "Come vuoi filtrare?", a dropdown menu showing "*****", and a blue button labeled "Rimuovi filtri". To the right is a table with one row. The table header is "Esame della tipologia ematochimica". The body contains the text "Esame che verrà eseguito in data Venerdì 15 Settembre 2023 20:52" and "e non e' ancora stato prenotato". At the bottom of the row is a yellow button labeled "Modifica".

Figura 8: pagina senza filtri

The screenshot shows the filter section. It has the label "Come vuoi filtrare?" above a dropdown menu that currently shows "Stato". Below this is another label "Quale stato?" above a second dropdown menu that currently shows "Prenotato". At the bottom of this section is a blue button labeled "Filtra Esami".

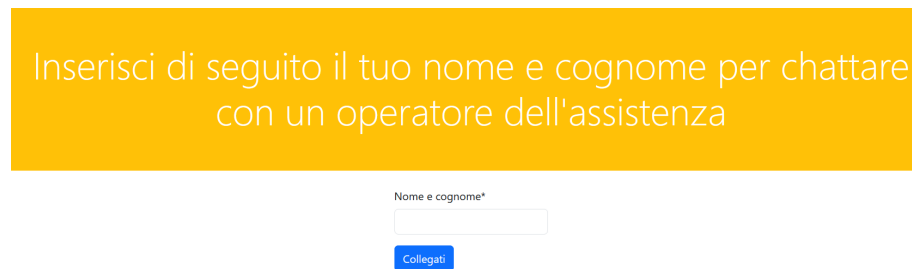
Figura 9: filtro stato

The screenshot shows the same green header "Stato attuale dei tuoi esami caricati". The filter section on the left is identical to Figure 9, with "Come vuoi filtrare?" set to "Stato" and "Quale stato?" set to "Prenotato", and the "Rimuovi filtri" button. The table on the right now has a header "Esame della tipologia allergologia". The body text reads "Esame che verrà eseguito in data Sabato 05 Agosto 2023 15:00" and "ed e' stato prenotato da provaž provaž". A yellow "Modifica" button is at the bottom of the row.

Figura 10: pagina filtrata

4.4 gestione_assistenza

In questa app vengono implementate le view e tutto il necessario per chattare con l'assistenza. Ho deciso di dividere in due view separate la richiesta di assistenza in quanto volevo renderla accessibile anche a chi non è registrato. Nella prima view controllo che l'utente che fa la richiesta sia registrato e se lo è creo una nuova chat e rediriziono l'utente subito alla pagina con la chat, altrimenti prima chiedo all'utente nome e cognome. La seconda view invece si occupa di controllare che l'utente possa accedere alla pagina (per esempio un utente che non fa parte dell'assistenza e che cerca di accedere ad una chat in attesa, viene rimandato alla home) e cambia lo stato della chat in base a chi accede. Per mostrare ai membri dell'assistenza clienti gli utenti in attesa viene usata una semplice *ListView* dove il queryset delle chat viene filtrato per mostrare solo quelle in attesa. Per la chat vera e propria ho usato *Djano-channels* ricalcando molto l'esempio delle slide dove ho usato l'id della chat come numero per il gruppo ma ho aggiunto uno stato ai messaggi scambiati dagli utenti per fare in modo che quando uno dei due utenti abbandona, l'altro venga notificato e non possa più mandare messaggi. Ho anche fatto una piccola aggiunta in javascript per permettere agli utenti di mandare messaggi premendo invio e non solo schiacciando un bottone. Come backend per i channel layers ho utilizzato gli in memory channel ma ho dovuto usare *Daphne* come server perché altrimenti con l'ultima versione di Django non funzionava correttamente e continuava ad utilizzare la versione normale del server al posto del server in grado di gestire le richieste ASGI.



Inserisci di seguito il tuo nome e cognome per chattare con un operatore dell'assistenza

Nome e cognome*

Collegati

Figura 11: Pagina richiesta assistenza



Figura 12: pagina con chat

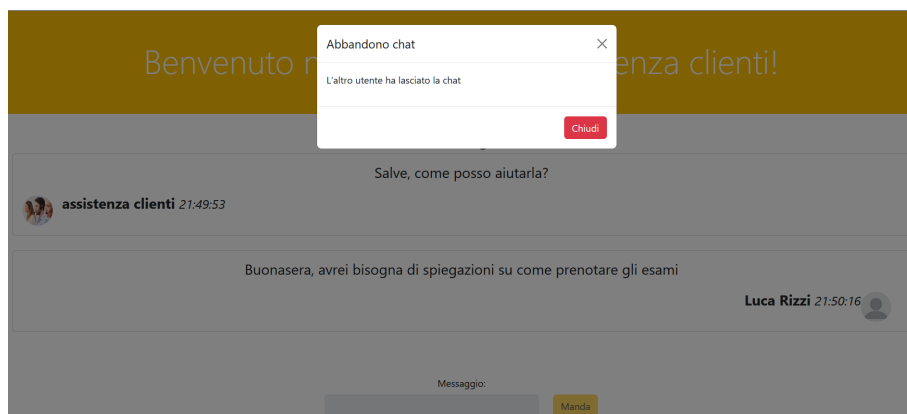


Figura 13: abbandono chat

4.5 autorizzazioni e gruppi

Per quanto riguarda le autorizzazione ho deciso di utilizzare due gruppi più il sistema di richiesta di login fornito da Django in quanto non voglio impedire ai medici o ai membri dell'assistenza di poter prenotare esami. Ho quindi creato un gruppo *Medici* a cui ho aggiunto i permessi di creare e modificare i modelli degli esami e dei medici. I permessi avrei potuto evitare di impostarli per questo gruppo visto che non devono accedere alla pagina admin e io nelle view controllo solo che un utente appartenga ad un certo gruppo ma ho pensato che in una situazione reale potrebbe essere comodo impostare anche i permessi perché potrebbero venire usati per un futuro aggiornamento. I membri del gruppo *Assistenza* invece possono accedere al pannello admin (infatti è da lì che creano i medici) e quindi è importante scegliere i giusti permessi: in generale ho dato in

permessi per vedere, creare e modificare tutti i modelli ad eccezione del modello delle chat perché sempre in un'ottica reale ho pensato che un membro potrebbe imbrogliare creando chat finte per nascondere il fatto di non stare lavorando. Inoltre non ho dato il permesso di eliminare utenti e medici perché ho pensato che un'azione così importante è meglio venga svolta da un admin o comunque qualcuno con un'autorità superiore. Nel progetto quindi oltre al decoratore *login_required* e al mixin *LoginRequiredMixin*, ho usato la libreria consigliata a lezione **django-braces** per gestire le appartenenze ai gruppi.

4.6 email

Per l'invio delle mail ho deciso di utilizzare il backend che stampa nel terminale le email inviate in quanto non avendo il sito dei dati reali, sarebbe difficile capire se funziona davvero utilizzando un smtp come gmail. Per lo scopo del progetto ho ritenuto fosse sufficiente mostrare che le mail vengono davvero mandate nel momento giusto e con il corretto contenuto. Per mandare le mail ho utilizzato la funzione *send_mail* fornita da Django. Nella pratica ho fatto innanzitutto un override del metodo *save* nel form di creazione degli utenti per mandare una mail all'indirizzo indicato subito dopo aver salvato l'utente nel database e poi un altro override del metodo *save* del modello degli esami per controllare se bisognasse mandare una mail per confermare la prenotazione oppure per notificare la modifica o cancellazione di un esame. Per riuscire a farlo ho dovuto utilizzare una libreria chiamata **django dirty-fields** che fornisce una classe che se ereditata da un modello permette di mostrare i campi che hanno subito modifiche e il loro vecchio valore. In questo modo potevo controllare per esempio lo stato per capire se era cambiato da 'disponibile' a 'prenotato' e mandare così una mail di conferma prenotazione.

4.7 Pagina admin

Per ogni modello non mi sono limitato a registrare il modello per farlo apparire nella pagina admin ma ho anche aggiunto certi campi visibili, ho aggiunto il filtraggio secondo alcuni parametri e la ricerca. La logica che ho seguito è stata quello di non includere campi con valori molto lunghi come per esempio il testo dei commenti per il modello *Commento* o la descrizione nel modello *Medico* in quanto la sezione con tutta la sequenza delle istanze di un certo modello penso debba poter essere attraversata a colpo d'occhio per capire se l'istanza che sto cercando è quella giusta. Per il filtraggio ho aggiunto solo campi con un numero predefinito di opzioni per evitare di avere una fila troppo lunga di opzioni e, se il modello la utilizzava, anche la data perché se aggiunta tra i campi filtro, non aggiunge tutte le date possibili ma un sistema più comodo con periodi prestabiliti come *Oggi* oppure *Quest'anno*. Per la ricerca invece ho permesso sempre di cercare tra nome e cognome associati agli utenti visto che in tutti i modelli c'è almeno un'associazione con il modello *Utente custom*. Quindi per esempio nel caso degli esami si può cercare il nome e/o cognome del medico che esegue l'esame oppure il nome e/o cognome del paziente. Nei modelli che

l'utilizzano, ho aggiunto anche la ricerca per data in quanto ho pensato che il filtro per data lo si usa quando non si ha una chiara idea della data associata all'istanza del modello (per esempio cerco un esame di un certo utente che so che è stato eseguito negli ultimi sette giorni ma non so di preciso quando) mentre la ricerca per data quando si è sicuri (per esempio cerco l'esame di un utente effettuato il giorno 25/05/2023).

5 Modelli

I modelli creati sono stati:

- Utente custom: un modello personalizzato per gli utenti;
- User Manager: un modello personalizzato di user manager per gestire il nuovo modello degli utenti;
- Medico: un modello collegato ad una istanza di utente per rappresentare i medici. Contiene la descrizione opzionale che possono aggiungere dopo essere stati aggiunti al sistema;
- Esame: un modello per rappresentare gli esami. Contiene i campi classici che ci si aspetta per l'esame come medico che l'esegue, paziente che lo prenota, tipologia di esame e data in cui verrà eseguito;
- Commento: un modello per rappresentare i commenti, ognuno dei quali collegato ad un medico;
- Chat: un modello per rappresentare le chat. Ha un solo collegamento con il modello utente e non due perché per permettere anche agli utenti non registrati di accedere all'assistenza, viene solo salvato il nome e cognome di chi la richiede.

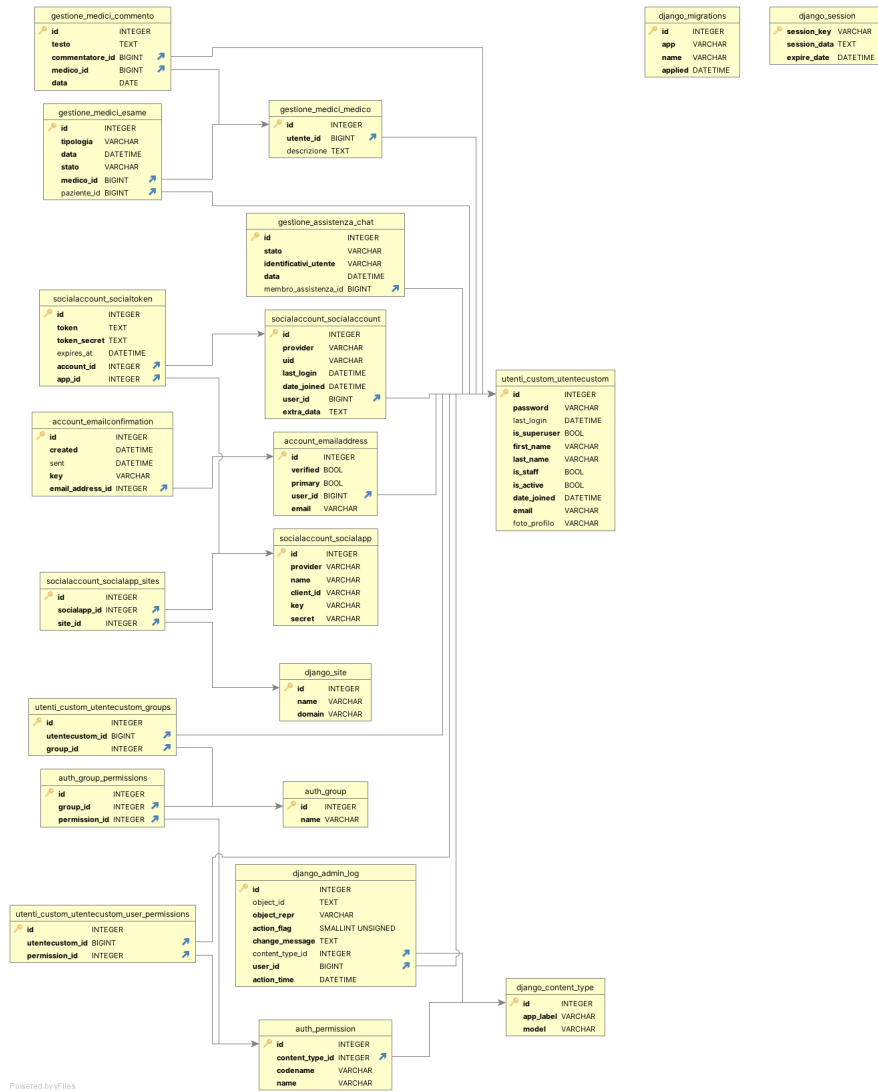


Figura 14: diagramma UML classi

6 Test

Per i test mi sono concentrato su due app: gestione_utenti e gestione_medici. Ho inizialmente creato un paio di test sulla view *situazione_utente* per prendere confidenza con i vari metodi e la logica che si usa per creare i test. In questi due test ho verificato che il sito redirezioni l'utente non loggato verso la giusta pagina nel caso voglia vedere il suo archivio degli esami e poi ho verificato che

nel caso ci siano gli esami di più utenti nel database, vengano mostrati solo gli esami che appartengono all'utente che accede alla pagina. I veri test li ho fatti sulla view per la ricerca degli esami e ho testato:

- se non vengono inseriti parametri allora devono venire mostrati tutti gli esami disponibili;
- per ciascun parametro che può essere ricercato, ho controllato che se viene inserito allora gli esami mostrati devono rispettarlo;
- il numero di esami mostrati per pagina deve essere quello corretto specificato dalla variabile *paginate_by*;
- che vengano mostrati solo gli esami disponibili e non anche quelli cancellati, prenotati o eseguiti;
- se non ci sono esami disponibili, allora non devono essere mostrati esami e deve essere presente la scritta *Non sono stati trovati esami*;
- che venga mostrato il giusto contenuto per ogni esame, in particolare tutti i vari campi che lo compongono e un bottone per prenotare l'esame.

Dell'app gestione_medici ho invece testato la spedizione della mail che viene effettuata dal metodo *save* del modello *Esame*. Ho creato tre test:

- un test per verificare che quando un utente prenota una visita, gli venga mandata una mail di conferma;
- un test per verificare che quando viene modificato un campo dell'esame, venga mandata una mail all'utente di notifica;
- un test per verificare che quando un utente cancella un'esame venga mandata una mail all'utente di notifica.

In ogni test delle mail viene anche testato che il testo mandato sia quello corretto.

7 Schermate aggiuntive

Inserisco un altro paio di schermate del sito secondo me interessanti.

31 persone totali in attesa, 5 in questa pagina		
Identificativi utente	Da quando e' in attesa	Accesso
prova2 prova2	Martedì 13 Giugno 2023 23:10	Entra
prova2 prova2	Martedì 13 Giugno 2023 23:12	Entra
prova2 prova2	Martedì 13 Giugno 2023 23:20	Entra
prova2 prova2	Martedì 13 Giugno 2023 23:26	Entra
prova2 prova2	Martedì 13 Giugno 2023 23:26	Entra

[« prima](#)
[precedente](#)
[Pagina 1 di 7](#)
[prossima](#)
[ultima »](#)

Figura 15: Pagina con gli utenti in attesa di assistenza



Presentazione di Luigi Bianchi

Sono un medico laureato nel 2010 all'università di Modena e Reggio Emilia. Il mio punto di forza è il rapporto con i pazienti che cerco di curare nei miei dettagli.

Se sei stato un paziente del medico Luigi Bianchi e vuoi aggiungere un commento, clicca sul bottone sottostante

[Scrivi commento](#)

Ci sono 8 commenti, per vederli tutti clicca [qui](#), di seguito i più recenti:

Un medico fuori dal comune che consiglio a tutti.

 **rigoletta popovici** Pubblicato in data 05 Luglio 2023

Riesce a mettere a proprio agio anche una persona come me che ha sempre avuto paura di visite.

 **rossana vaccaro** Pubblicato in data 05 Luglio 2023

Non lo consiglio, sono arrivato in orario ma mi hanno fatto aspettare mezz'ora senza darmi spiegazioni.

 **finarosa Martini** Pubblicato in data 05 Luglio 2023

Figura 16: Pagina con la presentazione del medico e gli ultimi tre commenti

8 Problemi riscontrati

- Un primo problema che ho riscontrato è stata decidere come dividere il progetto in app in quanto non ero sicuro se fosse più comodo tenere le app unite e questo ha rallentato inizialmente lo sviluppo ma poi riflettendoci bene sono arrivato alla divisione finale in quattro app.
- Un altro problema è stato bootstrap in quanto ho deciso di usare l'ultima versione (5.3) ma negli esempi delle lezioni veniva usata la versione 4.6.1 e ho capito in seguito facendo ricerche che alcune classi erano state rimosse oppure cambiate ed era per quel motivo che le mie pagine erano diverse da quelle di esempio fatte in aula. Ho quindi dovuto cercare dei modi per ricreare quello che volevo in bootstrap 5.
- La pagina con esami caricati dai medici con il filtraggio è stata problematica inizialmente perché non sapevo bene come realizzarla visto che negli

esempi fatti a lezioni la parte di ricerca e quella in cui si mostrano i risultati erano separate. Inoltre volevo riuscire a fare funzionare tutto solo con richieste *GET* e non usando una *POST* e quindi non essendoci una netta differenza tra quando viene richiesta la pagina normale e quando viene richiesto di filtrare (se non per i parametri aggiuntivi nella querystring), ho dovuto scrivere una view che potesse funzionare in entrambi i casi. Ho poi avuto il problema della paginazione descritto nella sezione dell'app gestione_medici che mi ha richiesto un po' di tempo per capire il perché non funzionasse e soprattutto come risolvere. Il tutto cercando di non rendere troppo lunga e complicata la funzione (perché ho scelto di usare una function view), cosa che ho risolto spezzando certe parti in funzioni aggiuntive come per esempio il filtraggio vero e proprio del queryset.