



UNIVERSITÀ DEGLI STUDI DI CATANIA
DIPARTIMENTO DI MATEMATICA E INFORMATICA
CORSO DI LAUREA TRIENNALE IN INFORMATICA

Raciti Riccardo

Auction

RELAZIONE PROGETTO INGEGNERIA DEI
SISTEMI DISTRIBUITI

Professore Emiliano Alessio Tramontana
Professore Andrea Francesco Fornaia

Anno Accademico 2023 - 2024

Abstract

Questo progetto descrive il ciclo di sviluppo di un e-commerce basato su Django e PyCharm, inizialmente concepito per facilitare le aste online di oggetti unici. Nella fase iniziale, l'applicazione web è stata progettata con l'obiettivo di fornire una piattaforma intuitiva per la vendita e l'acquisto attraverso offerte dinamiche. PyCharm è stato utilizzato come ambiente di sviluppo integrato, accelerando il processo di sviluppo e migliorando la gestione del progetto.

Successivamente, il progetto ha subito un processo di refactoring per adottare un'architettura basata su microservizi. Questa fase ha permesso la modularità delle funzionalità del sistema, garantendo una maggiore scalabilità e semplificando la manutenzione del codice. Ogni microservizio, gestito in modo autonomo, si occupa di specifiche funzionalità, come autenticazione, gestione degli oggetti e monitoraggio delle offerte.

Indice

1	Introduzione	4
2	Implementazione Monolita	6
2.1	Gestione Utenti	7
2.1.1	Registrazione	8
2.1.1.1	Implementazione Registrazione	10
2.1.2	Log In	12
2.1.2.1	Implementazione Log In	14
2.1.3	Password Dimenticata	15
2.1.3.1	Implementazione Password Dimenticata	19
2.1.4	Profilo Utente	20
2.1.4.1	Implementazione Utente	21
2.1.5	Admin	22
2.1.5.1	Implementazione Admin	24
2.2	Gestione Item	25
2.2.1	Tutte le aste	25
2.2.2	Aste attive	27
2.2.3	Aste del Giorno	27
2.3	Implementazione Item	28
2.3.1	Offerta	31
2.3.1.1	Implementazione Offerta	33
3	Seperazione Responsabilità	38
3.1	Impostazioni Progetto	39
3.1.1	Configurazione dell'Applicazione	40
3.1.2	Configurazione del Database	42
3.1.3	Configurazione dell'Autenticazione e degli Utent	43
3.1.4	Configurazione dei Template	43
3.1.5	Configurazione del Linguaggio e della Zona Oraria	44
3.2	Modifiche nella funzionalità	45
3.2.1	Gestione Utenti	45

3.2.1.1	Implementazione CustomUser	45
3.2.1.2	Login con Email	47
3.2.1.3	Funzione per gli Staff User	48
3.2.2	Gestione Oggetti	49
3.2.3	Gestione Assistenza	50
4	Applicazione con microservizi	54
4.1	Gestione Utenti	57
4.1.1	Generazione Token	58
4.1.2	Chiamate Ajax	60
4.2	Gestione Oggetti	61
4.3	Classe Quest	61
4.3.1	Definizione View	62
4.3.2	Chiamate Ajax	64
4.4	Gestione Assistenza	65
5	UML	66
5.1	Gestione Utenti	66
5.1.1	CustomUser	66
5.1.2	Registrazione e Login	67
5.2	Gestione Oggetti	68
5.2.1	Definizioni Classi	68
5.2.2	View	68
5.3	Gestione Assistenza	69
5.3.1	Definizioni Classi	69
5.3.2	View	70
	Conclusione	71
	Bibliografia	72

Capitolo 1

Introduzione

Il progetto in questione rappresenta un viaggio entusiasmante attraverso lo sviluppo di un e-commerce innovativo, inizialmente ideato con l'obiettivo di facilitare la vendita di oggetti unici attraverso aste online. Utilizzando il framework Django [1] e l'ambiente di sviluppo PyCharm, è stato creato un'applicazione web robusta e intuitiva per connettere acquirenti e venditori in un'esperienza di shopping all'avanguardia.

Fase Iniziale: E-commerce, Aste Online e Gestione User

La fase iniziale del progetto ha visto la creazione di un e-commerce basato su Django, progettato per ospitare aste online. L'obiettivo principale era fornire una piattaforma versatile in grado di gestire la vendita di oggetti unici attraverso un sistema di offerte dinamiche. PyCharm è stato il nostro compagno ideale durante questa fase, offrendo un ambiente di sviluppo integrato che ha accelerato il processo di scrittura del codice e facilitato la gestione del progetto.

Le funzionalità di base includevano la registrazione degli utenti, la gestione degli oggetti in vendita, il monitoraggio delle offerte e la conclusione delle aste. L'interfaccia utente è stata progettata per garantire un'esperienza utente fluida e coinvolgente, favorendo la partecipazione attiva degli utenti sia come venditori che acquirenti.

Fase Successiva: Refactoring e Introduzione dei Microservizi

Dopo la fase iniziale, è stata effettuata una fase di refactoring a favore di un approccio basato su microservizi. Il refactoring è stato implementato per separare le diverse funzionalità del sistema in servizi modulari e autonomi. Questo approccio ha portato a una maggiore scalabilità, facilitando la gestione e la manutenzione del codice.

Ogni microservizio è stato progettato per gestire specifiche funzionalità del sistema, come autenticazione, gestione degli oggetti, monitoraggio delle offerte e così via. PyCharm è stato ancora una volta fondamentale per la fase

di refactoring, fornendo strumenti avanzati per la navigazione del codice e il debugging, garantendo una transizione senza intoppi verso un'architettura basata su microservizi.

Capitolo 2

Implementazione Monolita

In questa fase è stato implementato il progetto come un'unica applicazione che gestisse tutte le attività e funzione del sito web.

All'interno del sito web la pagina principale è la seguente

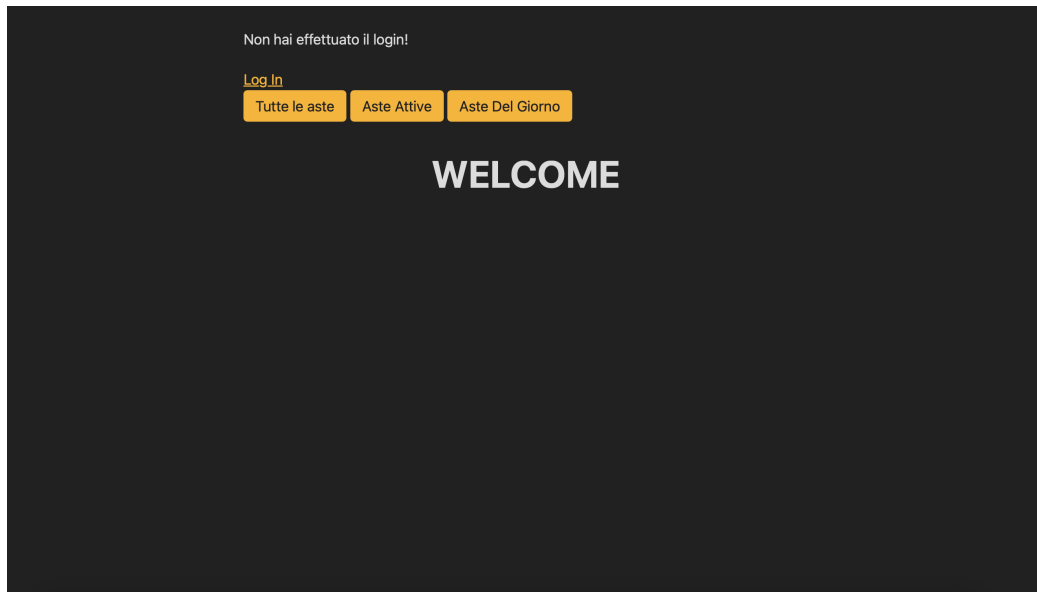
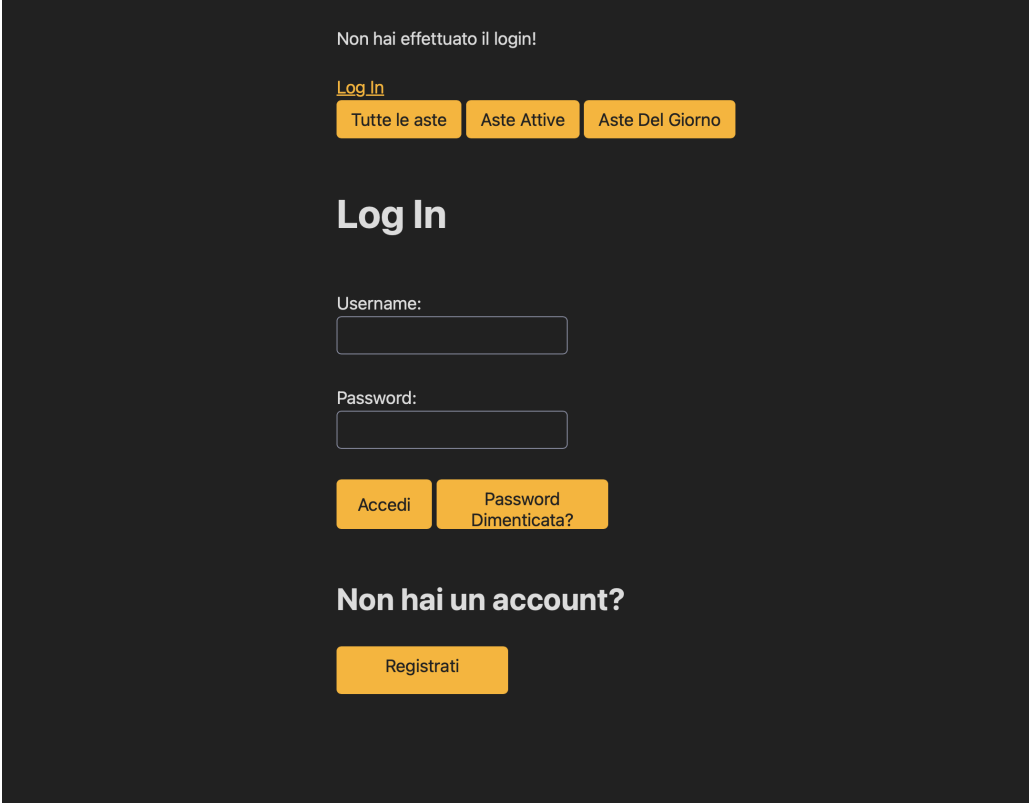


Figura 2.1: Schermata iniziale.

Nell'immagine vi è presente la scritta **WELCOME** e lo spazio sottostante è ideato per l'aggiunta di un logo o immagine del sito. I tre pulsanti presenti verranno trattati nella sezione specifica.

2.1 Gestione Utenti

Come si vede nell'immagine 2.1 in alto compare la scritta "Non hai effettuato il login!" sotto la seguente frase vi è scritto *Log In*. Cliccando sul link si aprirà la seguente pagina.



Non hai effettuato il login!

[Log In](#)

Tutte le aste Aste Attive Aste Del Giorno

Log In

Username:

Password:

Accedi Password Dimenticata?

Non hai un account?

Registrati

Figura 2.2: Pagina LogIn.

2.1.1 Registrazione

Se l'utente non ha effettuato la registrazione può cliccare il pulsante **REGISTRAZIONE**, Fig. 2.2.

Una volta cliccato sul seguente bottone l'utente verrà reindirizzato nella seguente pagina

Non hai effettuato il login!

[Log In](#)

[Tutte le aste](#) [Aste Attive](#) [Aste Del Giorno](#)

Sign up

Email:
 Obbligatorio. Inserisci un indirizzo email valido.

Username:
 Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

First name:
 Il tuo Nome

Last name:
 Il tuo Cognome

Password:

- 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:
 Enter the same password as before, for verification.

[Registrati](#)

[Home](#)

Figura 2.3: Schermata Registrazione.

2.1.1.1 Implementazione Registrazione

Questa pagina è generata dalla **view** `SignUpView`

```
from django.urls import reverse_lazy
from django.views import generic
from django.shortcuts import redirect
from .form import CustomUserCreationForm
from django.contrib.auth.models import Group
from django.contrib.auth import login, authenticate

2 usages  ▶ Riccardo Raciti

class SignUpView(generic.CreateView):
    form_class = CustomUserCreationForm
    success_url = reverse_lazy("login")
    template_name = "Registration/signup.html"

▶ Riccardo Raciti
def form_valid(self, form):
    response = super().form_valid(form)
    email = form.cleaned_data.get('email')
    password = form.cleaned_data.get('password1')
    username = form.cleaned_data.get('username')

    user = authenticate(self.request, username=username, password=password)

    if user is not None:
        login(self.request, user)

        # Aggiungi l'utente al gruppo "Acquirenti"
        acquirenti_group = Group.objects.get(name='Acquirenti')
        user.groups.add(acquirenti_group)

        # Reindirizza alla vista principale
        return redirect('all')

    return super().form_invalid(form)
```

Figura 2.4: View `SignUpView`.

La seguente classe utilizza una classe esportata dalla cartella **form** e esporta la classe **CustomUserCreationForm**

```
from django import forms
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth.models import User

2 usages  ▲ Riccardo Raciti
class CustomUserCreationForm(UserCreationForm):
    email = forms.EmailField(max_length=254, help_text='Obbligatorio. Inserisci un indirizzo email valido.')
    first_name = forms.CharField(max_length=30, required=True, help_text='Il tuo Nome')
    last_name = forms.CharField(max_length=30, required=True, help_text='Il tuo Cognome')

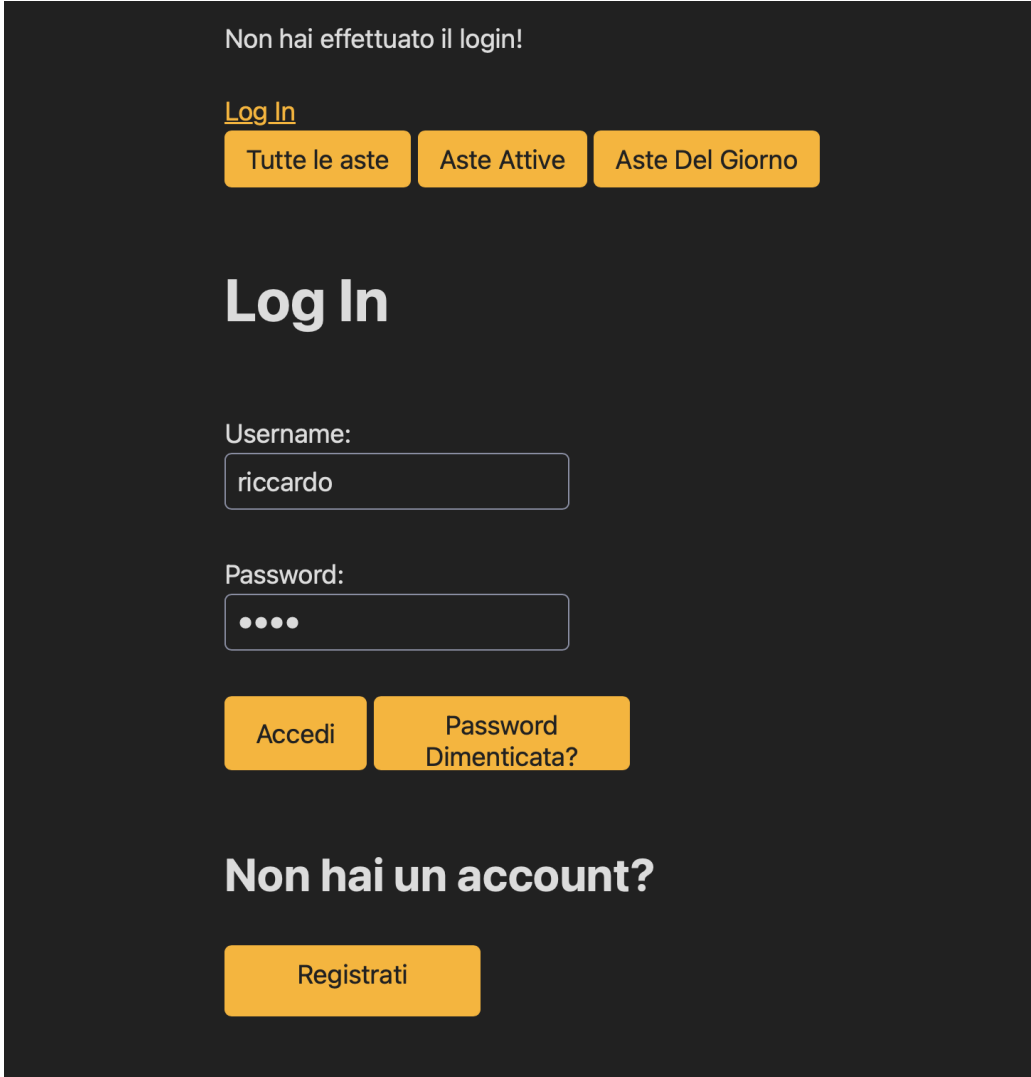
▲ Riccardo Raciti
class Meta:
    proxy = True
    model = User
    fields = ('email', 'username', 'first_name', 'last_name', 'password1', 'password2')
```

Figura 2.5: View SignUpView.

Questa classe permette di creare gli utenti e selezionare i parametri mostrati nella figura 2.3, poichè estende il modello **User** di Django, il quale prevede solo *username* e *password*. Inoltre viene implementato il **PROXY**.

2.1.2 Log In

Nella fase di login l'utente deve inserire **username** e **password**, ovviamente nel caso in cui abbia effettuato la fase di registrazione.



Non hai effettuato il login!

[Log In](#)

Tutte le aste Aste Attive Aste Del Giorno

Log In

Username:

Password:

Accedi Password Dimenticata?

Non hai un account?

Registrati

Figura 2.6: Pagina LogIn, compilazione campi.

Una volta effettuato il login l'utente verrà reindirizzato nella schermata iniziale.

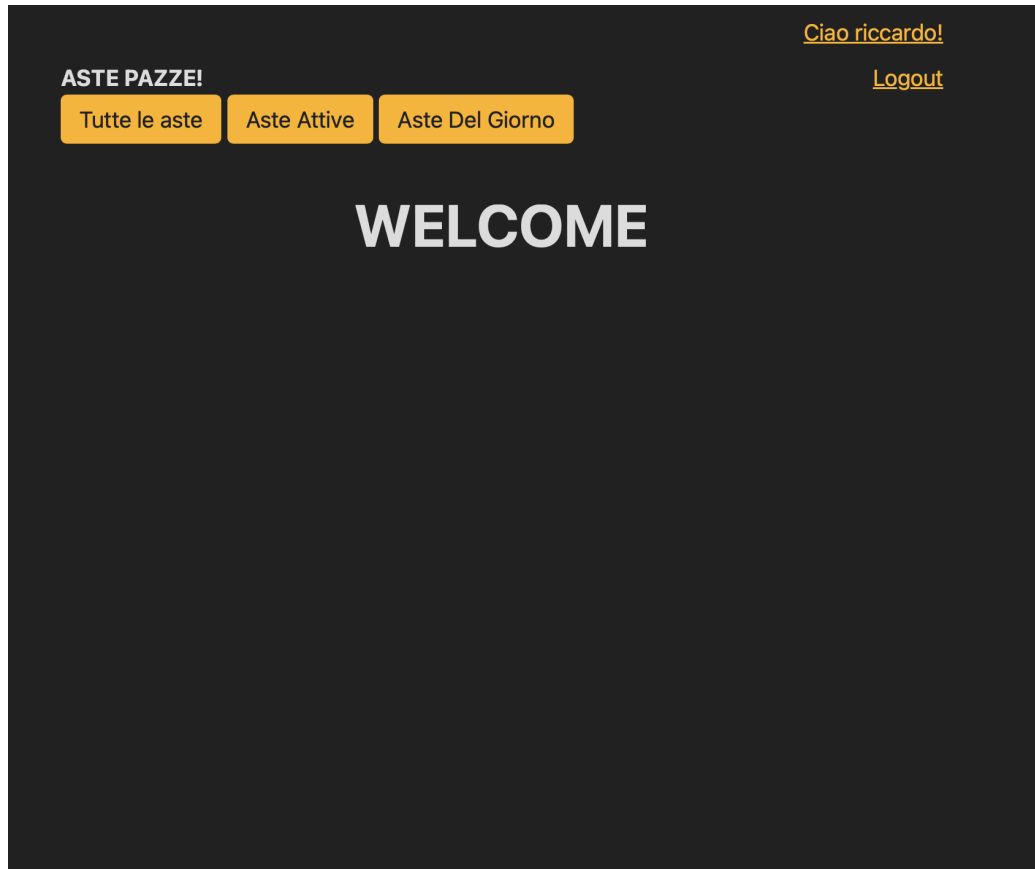


Figura 2.7: Home page dopo il login.

Una volta effettuato il login comparirà nella parte superiore, dove prima era presente la scritta "non hai effettuato il login", **ASTE PAZZE**, che denota il nome del sito web. Mentre sulla destra è presente un saluto seguito dal nome dell'utente connesso e il tasto per effettuare il logout.

2.1.2.1 Implementazione Log In

Per l'implementazione della fase di Log In è stata estesa la classe **form** di Django. Che poi viene utilizzata da una view per la generazione del form.

```
from django.contrib.auth.forms import AuthenticationForm
2 usages  ↳ Riccardo Raciti
class LoginForm(AuthenticationForm):
    username = forms.CharField(label='Email / Username')
```

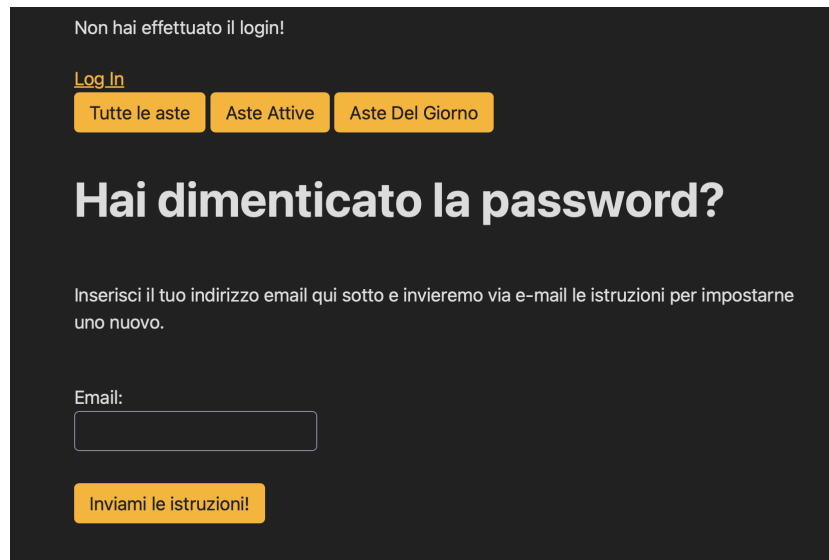
Figura 2.8: Classe LoginForm.

```
from .form import LoginForm
2 usages  ↳ Riccardo Raciti
class LoginView(auth_views.LoginView):
    form_class = LoginForm
    template_name = 'Registration/login.html'
```

Figura 2.9: LoginView.

2.1.3 Password Dimenticata

Nel caso in cui l'utente si è dimenticato la password può premere il pulsante **Password Dimenticata**, Fig .2.2. Così facendo l'utente verrà reindirizzato nella schermata:



Non hai effettuato il login!

[Log In](#)

Tutte le aste Aste Attive Aste Del Giorno

Hai dimenticato la password?

Inserisci il tuo indirizzo email qui sotto e invieremo via e-mail le istruzioni per impostarne uno nuovo.

Email:

Inviarmi le istruzioni!

Figura 2.10: Form password dimenticata.

Dove si potrà inserire l'email relativa all'account, una volta inserita l'e-mail e nel caso in cui corrisponda ad un'email valida comparirà la seguente schermata.

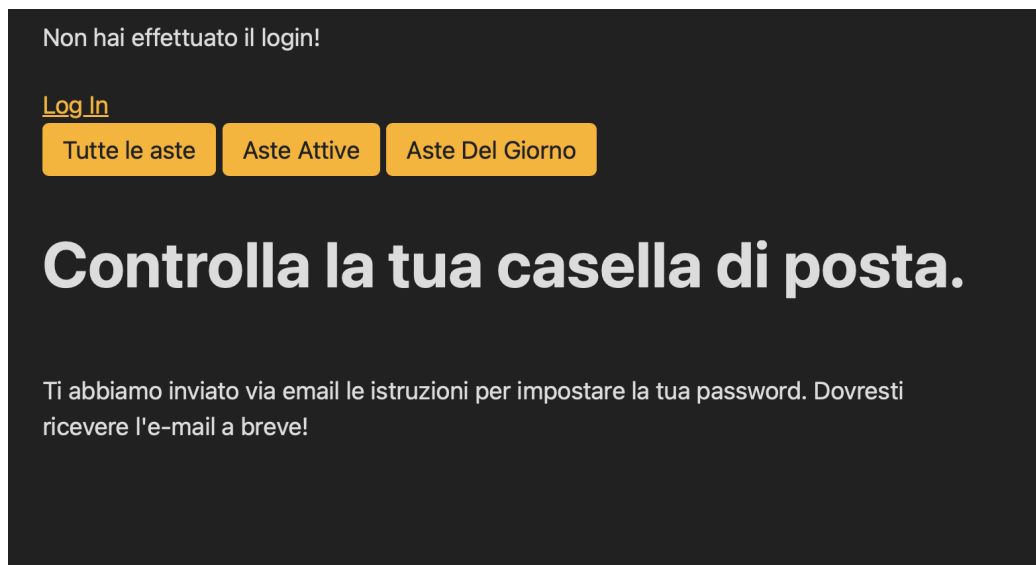


Figura 2.11: Conferma invio istruzione ridefinizione password.

PyCharm permette di simulare il funzionamento dell'email box e impostando il seguente parametro nel file *settings.py*

```
EMAIL_BACKEND = "django.core.mail.backends.filebased.EmailBackend"
EMAIL_FILE_PATH = BASE_DIR / "sent_emails"
```

Verrà creata una cartella chiamata "sent_emails" dove arriverà un messaggio del genere, il quale simula l'email mandata.

```
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 8bit
Subject: Password reset on 127.0.0.1:8000
From: webmaster@localhost
To: riccardo@tim.it
Date: Wed, 22 Nov 2023 10:52:48 -0000
Message-ID:
<170065036897.10987.7614582055411564009@mbp-di-riccardo.homenet.telecomitalia.it>

You're receiving this email because you requested a password reset for your user account at 127.0.0.1:8000.

Please go to the following page and choose a new password:

http://127.0.0.1:8000/accounts/reset/MQ/by2bk0-208fa35eadc943e2dee89a82722fe3df/

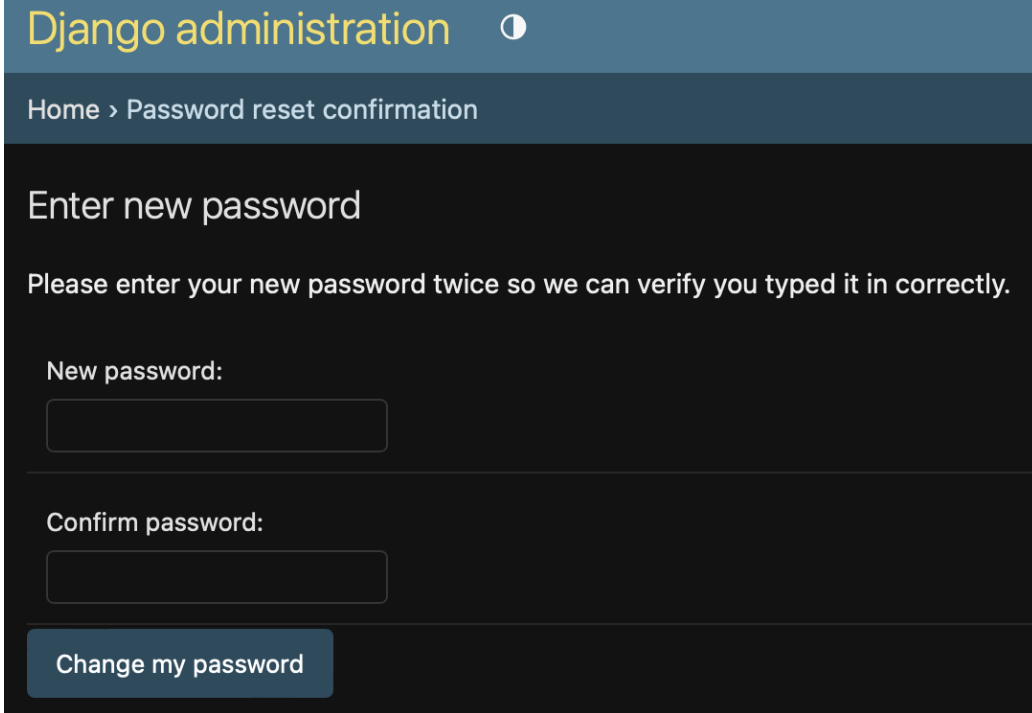
Your username, in case you've forgotten: riccardo

Thanks for using our site!

The 127.0.0.1:8000 team
```

Figura 2.12: Home page dopo il login.

Per eseguire il reset della password bisogna seguire la pagina *http*, dove comparirà



The screenshot shows the Django administration interface for password reset confirmation. At the top, the header reads "Django administration" in yellow text on a blue background, followed by a circular icon. Below this, a breadcrumb trail shows "Home > Password reset confirmation". The main content area has a dark background and contains the heading "Enter new password". Below the heading is a message: "Please enter your new password twice so we can verify you typed it in correctly." There are two input fields: the first is labeled "New password:" and the second is labeled "Confirm password:". At the bottom of the form is a blue button with the text "Change my password".

Figura 2.13: Pagina Reset Password Email.

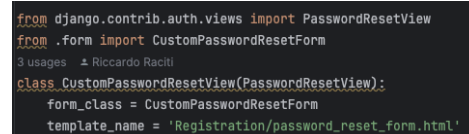
2.1.3.1 Implementazione Password Dimenticata

Per il reset delle password è stata estesa la classe **PasswordResetForm** di Django, che è stata poi utilizzata nella rispettiva view.



```
from django.contrib.auth.forms import PasswordResetForm
from django.core.mail import EmailMessage
class CustomPasswordResetForm(PasswordResetForm):
    email = forms.EmailField(label=_('Email'), help_text=_('Email'), widget=forms.EmailInput(attrs={'autocomplete': 'email'}))
```

Figura 2.14: Classe CustomPasswordResetForm.



```
from django.contrib.auth.views import PasswordResetView
from .form import CustomPasswordResetForm
3 usages  ± Riccardo Racioli
class CustomPasswordResetView(PasswordResetView):
    form_class = CustomPasswordResetForm
    template_name = 'Registration/password_reset_form.html'
```

Figura 2.15: CustomPasswordResetView.

Automaticamente dopo che si esegue il file html, cliccando sul tasto *Inviame le Istruzioni!*, Fig. 2.10, Django reindirizza l'utente al file html che genera la pagina Fig. 2.11, grazie all'aggiunta del seguente path nel file *urls.py* del progetto

```
path('password_reset_done/',
CustomPasswordResetView.as_view(),
name='password_reset_done')
```

Questo avviene poichè di default Django reindirizza gli utenti al path *password_reset_done/* e applicando una view si può customizzare la visione della pagina.

2.1.4 Profilo Utente

Se si clicca nel nome dell'utente loggato, Fig. 2.1, si viene reindirizzati nella pagina relativa all'utente.

Nella pagina si trova:

- Scritta di benvenuto;
- Nome e cognome dell'utente;
- Lista oggetti aggiudicati tramite l'asta;
- Se l'account è un account dello staff;
 - Nel caso in cui l'account è dello staff compare un link per la sezione admin del sito;
- Permessi dell'utente.

Di seguito sono riportati due differenti sezioni utente.

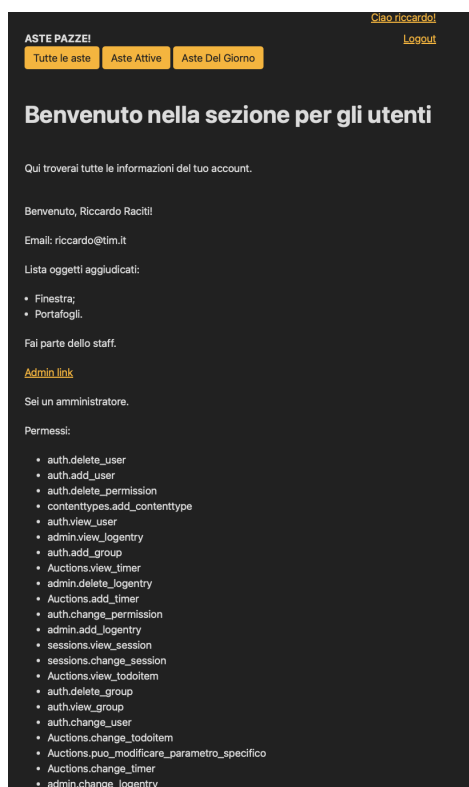


Figura 2.16: Profilo utente Admin.

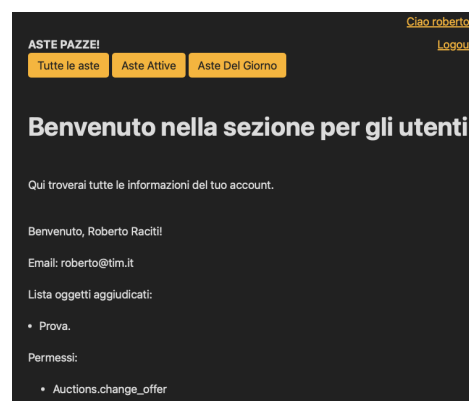


Figura 2.17: Profilo utente base.

2.1.5 Admin

Come mostrato nella figura 2.16, se l'utente fa parte dello staff può cliccare su un link per gestire il sito.

Una volta cliccato il link l'admin verrà reindirizzato nella pagina sottostante. In questa pagina è possibile gestire:

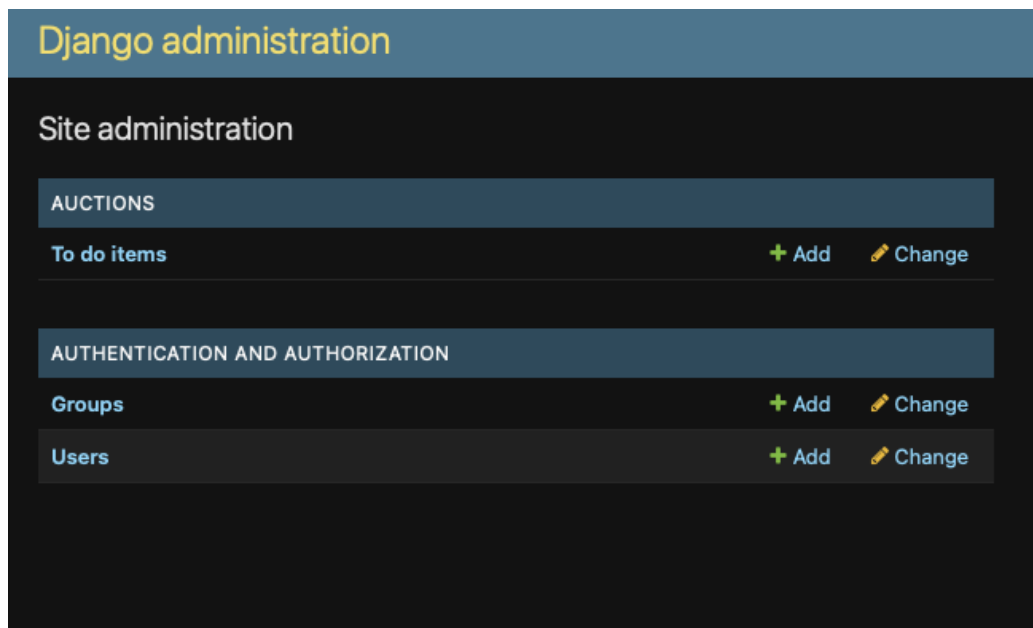
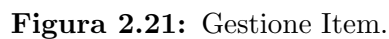


Figura 2.20: Pagina Admin.

- Elementi alla racconta **To Do items**, che racchiude gli elementi venduti dal sito 2.21;
 - Gli item si posso creare ed aggiungere;
 - Modificare un item già creato;
 - Eliminare un item presente.
- Gestire i gruppi, che nel nostro caso è solo uno *Acquirenti* 2.9;
- Gestire gli utenti 2.24.



2.1.5.1 Implementazione Admin

Come mostrato nel file html dell'utente, Fig. 2.18, se l'utente ha fa parte dello staff (*user.is_staff* viene reindirizzato nella pagina nativa di Django.

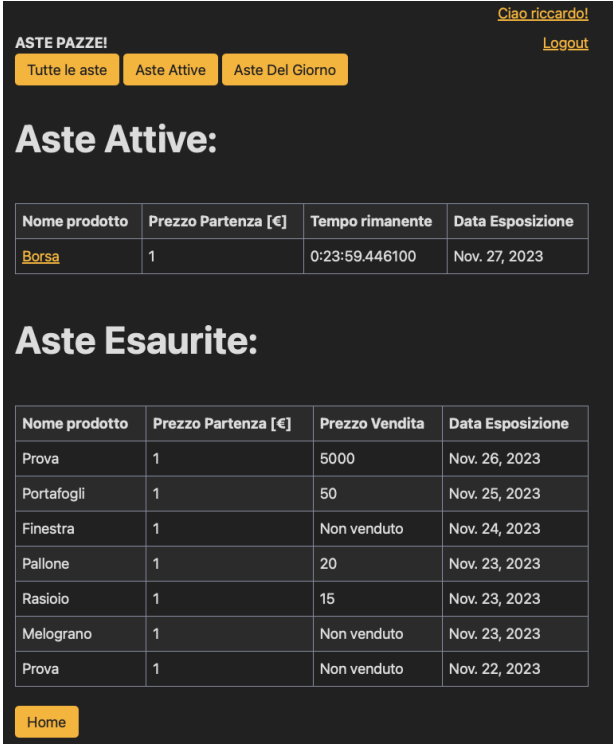
2.2 Gestione Item

Nella schermata principale che chiameremo *home*, Fig. 2.7, Possiamo vedere sotto il nome del sito tre bottoni

- Tutte le aste;
- Aste Attive;
- Aste del Giorno.

2.2.1 Tutte le aste

Nella seguente pagina vengono mostrate tutte le aste che sono state bandite all'interno sito, viene fatta una divisione in aste ancora attive e aste esaurite, la distinzione viene effettuata all'interno del file html utilizzando funzioni e attributi degli oggetti.



The screenshot shows a web interface with a dark background. At the top right, there is a user greeting 'Ciao riccardo!' and a 'Logout' link. Below this, the text 'ASTE PAZZE!' is displayed. Three orange buttons are visible: 'Tutte le aste' (selected), 'Aste Attive', and 'Aste Del Giorno'. The main content is divided into two sections: 'Aste Attive:' and 'Aste Esaurite:'. Each section contains a table of auction data.

Nome prodotto	Prezzo Partenza [€]	Tempo rimanente	Data Esposizione
Borsa	1	0:23:59.446100	Nov. 27, 2023

Nome prodotto	Prezzo Partenza [€]	Prezzo Vendita	Data Esposizione
Prova	1	5000	Nov. 26, 2023
Portafogli	1	50	Nov. 25, 2023
Finestra	1	Non venduto	Nov. 24, 2023
Pallone	1	20	Nov. 23, 2023
Rasioio	1	15	Nov. 23, 2023
Melograno	1	Non venduto	Nov. 23, 2023
Prova	1	Non venduto	Nov. 22, 2023

At the bottom left, there is a 'Home' button.

Figura 2.25: Schermata Tutte le Aste.

Nella tabella inerente le aste attive vengono mostrati nome prodotto, il prezzo di partenza dell'asta, il tempo rimanente e la data di esposizione. Mentre

nella tabella delle aste esaurite poiché il tempo è esaurito viene mostrato il prezzo di vendita o la scritta *Non venduto* nel caso in cui l'item non è stato venduto.

2.2.2 Aste attive

Nella seguente schermata viene mostrato il nome del prodotto, il prezzo di partenza, l'offerta attuale e il tempo rimanente.



Figura 2.26: Schermata Aste attive.

2.2.3 Aste del Giorno

In questa pagina come per Tutte le Aste vengono mostrate due tabelle e si differenzia da Tutte le Aste poichè in questa schermata ci saranno solo gli oggetti del giorno.

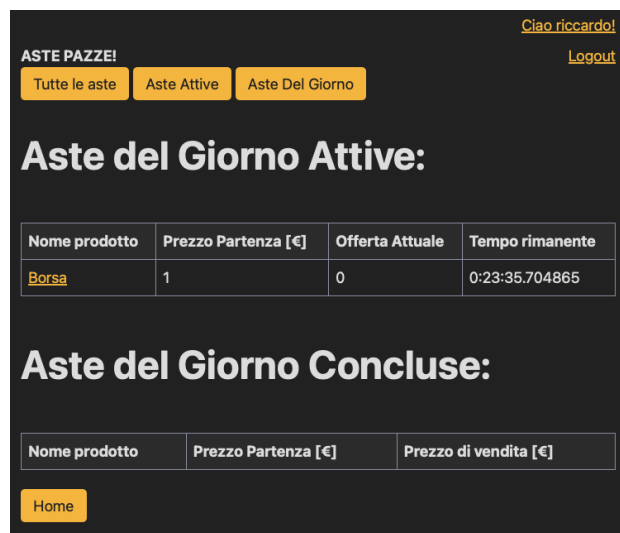


Figura 2.27: Schermata Aste del Giorno.

2.3 Implementazione Item

Gli item sono elementi contenuti nel database e sono definiti dalla classe **ToDoItem**, una classe estesa da *models.Model* di Django.

Le classi in Django sono utilizzate all'interno delle view, come visto prima.

Ad esempio per le pagine di visualizzazione viste nei paragrafi precedenti si utilizzano le seguenti view.

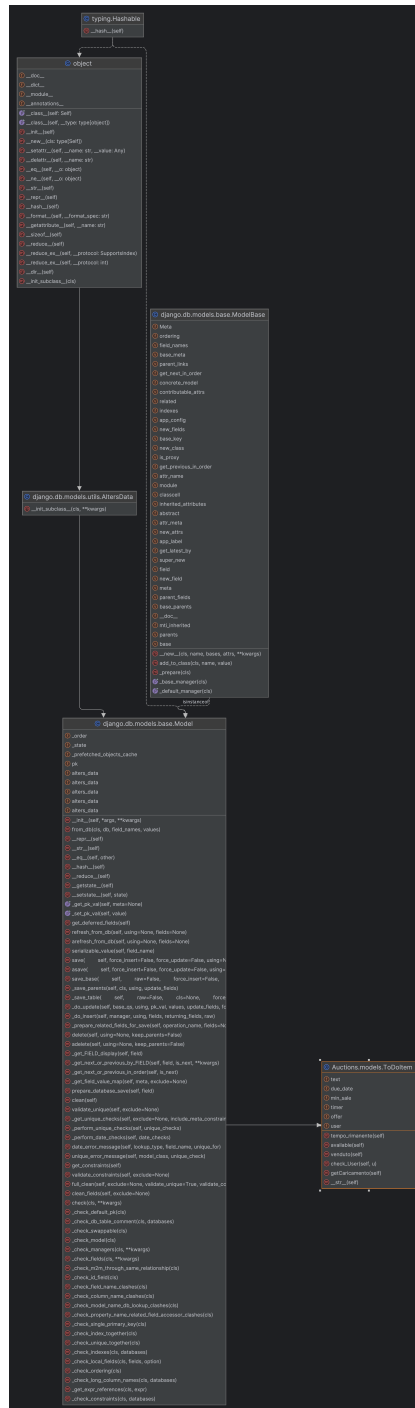


Figura 2.28: UML Model

```
class AllTodos(ListView):
    model = TodoItem
    template_name = "Auctions/index.html"

    1 Riccardo Raciti
    def get_queryset(self):
        return TodoItem.objects.all().order_by('-timer')
```

Figura 2.29: View per pagina Tutte le aste.

```
class ActiveTodos(ListView):
    model = TodoItem
    template_name = "Auctions/active.html"

    1 Riccardo Raciti
    def get_queryset(self):
        return TodoItem.objects.all().order_by('timer').order_by('-offer')
```

Figura 2.30: View per pagina Aste Attive.

```
class TodayTodos(ListView):
    model = TodoItem
    template_name = "Auctions/today.html"

    1 Riccardo Raciti
    def get_queryset(self):
        return TodoItem.objects.all().order_by('-timer')
```

Figura 2.31: View per pagina Aste del Giorno.

Tutte le view in questo caso impostano il model a **ToDoItem** per poi utilizzarlo nel *get_queryset*, in base alla logica cambiano gli *order_by* e il riferimento al template html corretto.

2.3.1 Offerta

Come mostrato nelle schermate precedenti gli item attivi sono cliccabili, un volta cliccato il nome dell'item l'app reindirizza l'utente nella schermata del dettaglio prodotto.



Figura 2.32: Schermata dettaglio prodotto.

Una volta giunti in quest schermata se l'utente è abilitato per poter effettuare un'offerta può cliccare il bottone *Fai un'offerta*, nel caso in cui l'utente non è abilitato gli verrà fornita una pagina vuota con scritto *Accesso negato effettua il login o registrati*.

Nel momento in cui si vuole fare un'offerta la schermata proposta è la seguente



Figura 2.33: Schermata Immissione offerta.

Le offerte vengono controllate, e i controlli svolti sono i seguenti:

- Controllo se l'offerta è maggiore del prezzo di partenza;
- Controllo se l'offerta è maggiore di quella attuale;
- Controllo se l'offerta è ammissibile in base al tempo rimanente.

Una volta effettuata l'offerta e tornati nella schermata desiderata si vedrà la tabella aggiornata.



Aste Attive:			
Nome prodotto	Prezzo Partenza [€]	Offerta attuale [€]	Tempo rimanente
<u>Borsa</u>	1	50	0:32:07.186363

Figura 2.34: Informazioni aggiornate dopo offerta.

2.3.1.1 Implementazione Offerta

Come detto precedentemente un utente può fare un'offerta se e solo se ha i permessi necessari per poterlo fare. Per la gestione dei permessi sono stati utilizzati i permessi associati ai gruppi, nell'applicazione per il momento è previsto un solo gruppo *Acquirenti* 2.23.

Come mostrato nella figura 2.4 quando un utente effettua la registrazione l'utente viene inserito nel gruppo *Acquirenti*. In questo modo quando un'utente effettua per la prima volta l'accesso all'app viene inserito automaticamente nel gruppo. Per far sì che gli utenti potessero modificare il campo **offer** degli item è stato creato un permesso specifico all'interno della classe *Acquirenti* del file *manage.py*.

```
def __permessi__(self):
    todoitem_content_type = ContentType.objects.
        get_for_model(ToDoItem)

    change_offer_permission = Permission.objects.
        get_or_create(
            codename='change_offer',
            content_type=todoitem_content_type,
            name="Puo modificare il prezzo dell'offerta"
        )[0]

    self.group.permissions.add(
        change_offer_permission)
```

L'UML relativo all'intera classe *Acquirenti* è fornito nella pagina successiva.

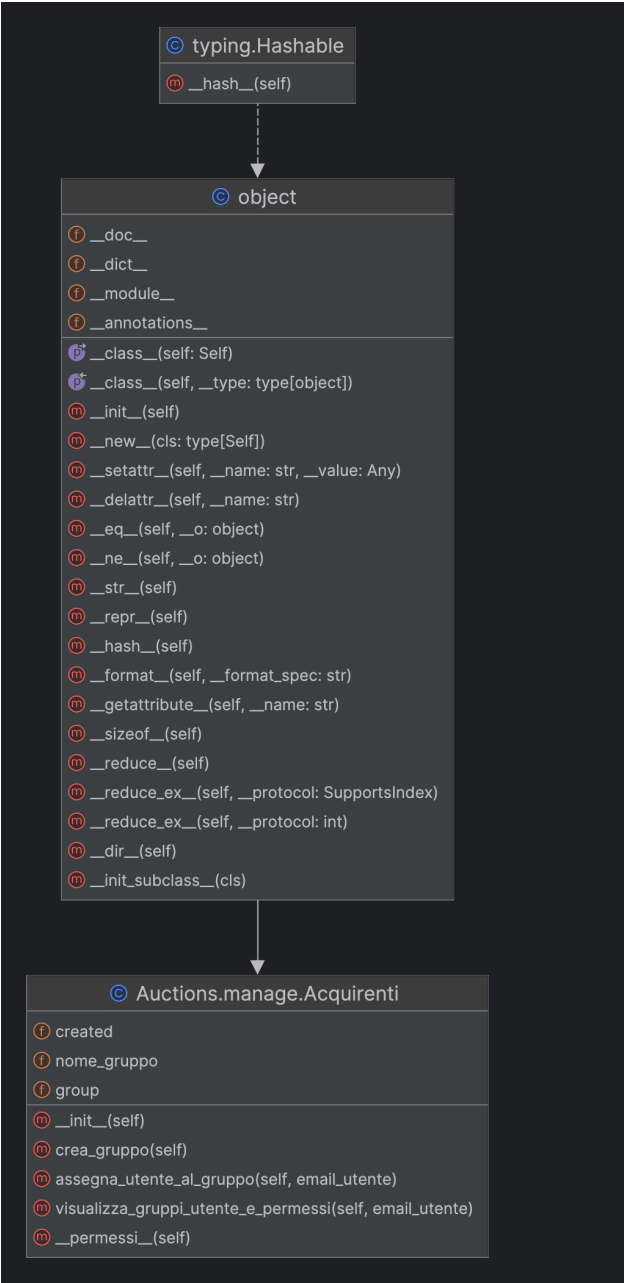


Figura 2.35: UML Acquirenti.

Tuttavia questo non basta per permettere ai soli utenti di effettuare le offerte. Per permettere solo agli Acquirenti, quindi coloro che hanno il permesso di modificare offer, è stato utilizzato il **design patterns *Decorator***. Quindi per applicare il design patter alla modifica dell'offerta è stato aggiunta questa linea di codice sopra la view che permette la modifica dell'offerta.

```
@method_decorator(GroupRequiredMixin('Acquirenti'),  
    name='dispatch')  
class ModificaOfferView(View):  
    ...
```

All'interno di *ModificaOfferView* viene definita la funzione post per la modifica del campo tramite post, funzione post nella pagina successiva.

```
def post(self, request, todoitem_id):
    todoitem = get_object_or_404(ToDoItem, pk=
        todoitem_id)

    new_offer_value = request.POST.get('new_offer')

    if float(new_offer_value) < todoitem.min_sale:
        messages.error(request, "L'offerta deve
            essere maggiore del prezzo di partenza.")
        return render(request, self.template_name, {
            'todoitem': todoitem})

    if float(new_offer_value) <= todoitem.offer:
        messages.error(request, "L'offerta deve
            essere maggiore di quella attuale.")
        return render(request, self.template_name, {
            'todoitem': todoitem})

    if not todoitem.available():
        messages.error(request, "Il tempo per fare
            offerte e' scaduto.")
        return render(request, self.template_name, {
            'todoitem': todoitem})

    current_user = request.user
    todoitem.offer = new_offer_value
    todoitem.user = current_user
    todoitem.save()

    return render(request, self.template_name, {'
        todoitem': todoitem})
```

Dove gli if implementano i controlli discussi precedentemente.

Una cosa importante è notare che dopo che l'offerta viene accettata viene aggiornato il campo *todoitem.user*, che indica l'acquirente che ha effettuato l'offerta maggiore per il relativo item.

Osservando il codice UML seguente si può vedere come ToDoItem esporti i permessi e gruppi dagli utenti.

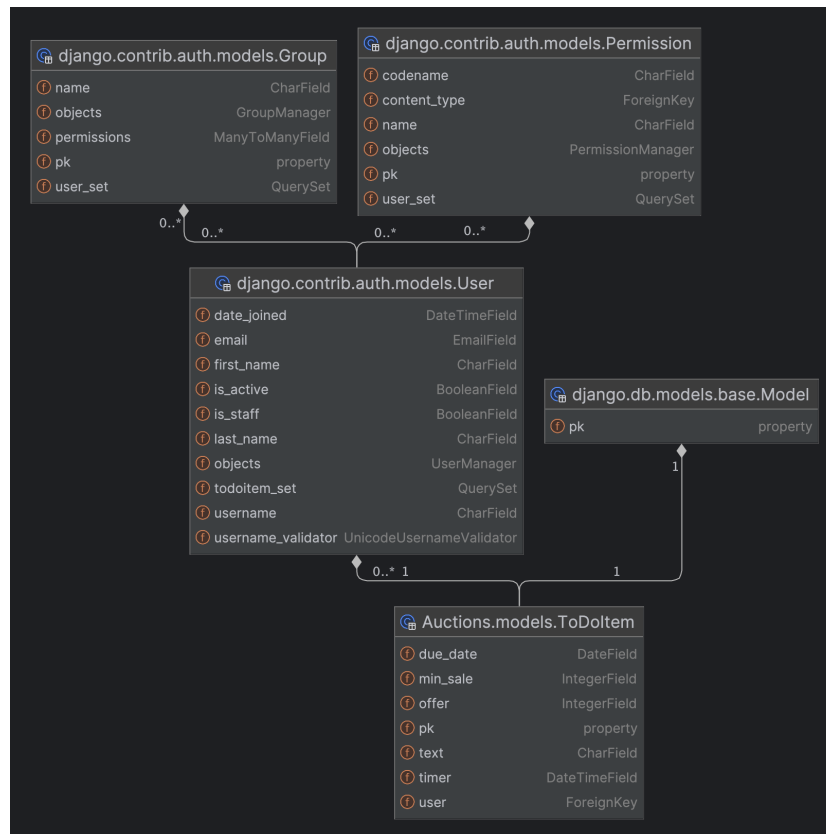


Figura 2.36: UML Dettaglio Prodotto.

Capitolo 3

Seperazione Responsabilità

In questo capitolo verrà mostrato il refactoring dell'applicazione precedentemente esposta, utilizzando anche in questo caso **Django**.

Nell'applicazione seguente viene creato un progetto che prende il nome di ***Auctionsv2*** dove al suo interno saranno presenti tre applicazioni che svolgeranno il loro compito come microserizi, si noti che è stata aggiunta una funzionalità in più nella versione seguente dell'applicazione.

Le applicazioni create sono:

- *Gestione Utenti*, questo microservizio si occuperà di gestire tutte le operazioni per la registrazione, login, vista informazioni degli utenti;
- *Gestione Oggetti*, questo microservizio si occuperà di gestire le operazioni di creazione, visualizzazione, assegnazione degli oggetti;
- *Gestione Assistenza*, questo microservizio (non presente nella versione precedente) si occuperà di una sezione offerta agli utenti dove sarà possibile effettuare domande per l'assistenza ai gestori del sito.

3.1 Impostazioni Progetto

Per prima cosa è importante definire correttamente le modifiche effettuate al file *settings.py* del progetto.

Il file *settings.py* è uno dei file chiave in un progetto Django. Contiene molte configurazioni che definiscono il comportamento globale dell'applicazione. Alcuni degli aspetti principali che il file *settings.py* gestisce includono:

- **Configurazione dell'Applicazione:** Elenco delle applicazioni installate nel progetto Django. Ogni applicazione ha il proprio set di funzionalità e può essere inclusa o esclusa dal progetto;
- **Configurazione del Database:** Specifica le impostazioni per la connessione al database, come il motore del database, il nome del database, l'utente e la password;
- **Configurazione dell'Autenticazione e degli Utenti:** Specifica il modello di utente utilizzato, le configurazioni per l'autenticazione, le impostazioni di registrazione e così via;
- **Configurazione dei Middleware:** I middleware sono componenti che elaborano le richieste HTTP prima che raggiungano le viste. Questi possono essere utilizzati per eseguire azioni come l'autenticazione, la gestione delle sessioni, la compressione del contenuto, ecc;
- **Configurazione dei Template:** Impostazioni relative ai motori di template, ai percorsi dei template e ad altre opzioni legate alla gestione delle pagine HTML;
- **Configurazione del Linguaggio e della Zona Oraria:** Definisce la lingua predefinita e la zona oraria del progetto.

Nei paragrafi successivi verrà mostrato come vengono configurati i vari campi.

3.1.1 Configurazione dell'Applicazione

Come espresso precedentemente il progetto seguente presenta tre applicazioni.

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'rest_framework',  
    'gestione_utenti.apps.GestioneUtentiConfig',  
    'gestione_oggetti.apps.GestioneOggettiConfig',  
    'gestione_assistenza.apps.GestioneAssistenzaConfig'  
]
```

Figura 3.1: Settings applicazioni.

Gli ultimi tre indicano le applicazioni aggiunte.

Di seguito vengono riportati i codici esportati nel file settings.

```
from django.apps import AppConfig

1 usage
class GestioneUtentiConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'gestione_utenti'
```

Figura 3.2: File app gestione_utenti.

```
from django.apps import AppConfig

1 usage
class GestioneOggettiConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'gestione_oggetti'
```

Figura 3.3: File app gestione_oggetti.

```
from django.apps import AppConfig

1 usage
class GestioneAssistenzaConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'gestione_assistenza'
```

Figura 3.4: File app gestione_assistenza.

3.1.2 Configurazione del Database

In visione della partizione totale delle responsabilità, è stato utilizzato un solo database ma al cui interno ogni app definisce una tabella totalmente scollegata dalle altre.

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.sqlite3',  
        'NAME': BASE_DIR / 'db.sqlite3',  
    }  
}
```

Figura 3.5: Settings Database.

Come mostrato nella figura sottostante si può osservare la struttura delle tabelle.



Amministrazione sito	
AUTENTICAZIONE E AUTORIZZAZIONE	
Gruppi	+ Aggiungi Modifica
GESTIONE_ASSISTENZA	
Messages	+ Aggiungi Modifica
GESTIONE_OGGETTI	
Items	+ Aggiungi Modifica
GESTIONE_UTENTI	
Utenti	+ Aggiungi Modifica

Figura 3.6: Struttura Database.

Ogni applicazione ha la sua tabella, nella seguente versione ogni applicazione genera una sola tabella.

3.1.3 Configurazione dell'Autenticazione e degli Utenti

In questa applicazione è stato creato un modulo specifico per gli utenti che viene ereditato da *AbstractUser*.

Per poter permettere l'utilizzo di questo oggetto **User** specifico bisogna aggiungere il seguente codice al file *settings*

```
AUTH_USER_MODEL = 'gestione_utenti.CustomUser'
AUTHENTICATION_BACKENDS = ['gestione_utenti.backends.EmailBackend']

LOGIN_REDIRECT_URL = "home"
LOGOUT_REDIRECT_URL = "home"

EMAIL_BACKEND = "django.core.mail.backends.filebased.EmailBackend"
EMAIL_FILE_PATH = BASE_DIR / "sent_emails"
```

Figura 3.7: Settings Custom User.

Le prime due righe definiscono il modello dell'utente da utilizzare, la terza e quarta riga definiscono i template da mostrare dopo la fase di login o logout e le ultime righe (come per la versione dell'applicazione precedente) servono per definire la simulazione dell'invio dei messaggi.

3.1.4 Configurazione dei Template

Ogni applicazione avrà la sua cartella templates, ma verrà utilizzata pure la cartella templates del progetto contenitore per modificare i template di base dell'applicazione.

3.1.5 Configurazione del Linguaggio e della Zona Oraria

Il linguaggio e la zona orarie sono state modificate per il miglior funzionamento del database.

```
LANGUAGE_CODE = 'it-IT'  
TIME_ZONE = 'Europe/Rome'  
  
USE_I18N = True  
  
USE_TZ = True
```

Figura 3.8: Settings linguaggio e zona oraria.

3.2 Modifiche nella funzionalità

In questo paragrafo verranno mostrate le modifiche apportate al progetto rispetto la versione precedente.

3.2.1 Gestione Utenti

Nella gestione utenti sono presenti tre modifiche:

- Definizione Custom User;
- Login effettuabile tramite email;
- Funzionalità aggiuntiva per gli *staff user*.

3.2.1.1 Implementazione CustomUser

Come mostrato nella figura 3.7 vengono utilizzati due nuove classi *CustomUser* e *EmailBackend*.

CustomUser è una classe che viene estesa da *AbstractUser*

```
class CustomUser(AbstractUser):
    email = models.EmailField(_('email address'),
                               unique=True)

    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)

    objects = CustomUserManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['username']

    def __str__(self):
        return self.email
```

La seguente classe per poter essere utilizzata correttamente ed evitare conflitti utilizza l'oggetto *CustomUserManager()*

```
class CustomUserManager(BaseUserManager):
    def create_user(self, email, username, password=
        None, **extra_fields):
        if not email:
            raise ValueError('The Email field must
                be set')
        email = self.normalize_email(email)
        user = self.model(email=email, username=
            username, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_superuser(self, email, username,
        password=None, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True
        )
        return self.create_user(email, username,
            password, **extra_fields)
```

Per poter vedere la tabella degli utenti all'interno del sito per la gestione di Django bisogna aggiungere il seguente codice nel file *admin.py*

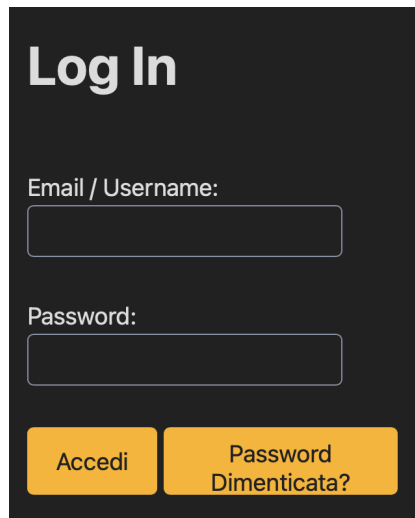
```
User = get_user_model()

class UserAdmin(BaseUserAdmin):
    add_form = RegisterForm
    form = RegisterForm
    model = User
    list_display = ['email', 'username', 'first_name',
                   'last_name', 'is_active', 'is_staff']

admin.site.register(User, UserAdmin)
```

3.2.1.2 Login con Email

Con le modifiche apportate alla classe User è possibile effettuare il login tramite l'utilizzo dell'email



The image shows a dark-themed login form. At the top, the text "Log In" is displayed in a large, white, sans-serif font. Below this, there are two input fields. The first is labeled "Email / Username:" and the second is labeled "Password:". Both labels are in a small, white, sans-serif font. The input fields are white with rounded corners and a thin white border. At the bottom of the form, there are two yellow buttons with rounded corners. The left button is labeled "Accedi" in a black, sans-serif font. The right button is labeled "Password Dimenticata?" in a black, sans-serif font.

Figura 3.9: Login con email.

3.2.1.3 Funzione per gli Staff User

Nella schermata dell'utente, se l'utente fa parte dello staff comparirà il seguente link

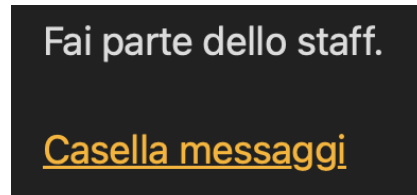


Figura 3.10: Link Messaggi.

Il quale link porterà alla pagina contenente tutti i messaggi mandati da parte degli utenti, con email dell'utente che ha mandato il messaggio, il messaggio scritto e la data in cui è stata effettuata la domanda.

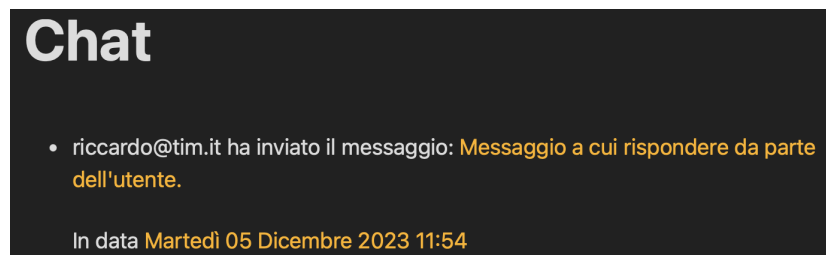


Figura 3.11: Lista messaggi.

3.2.2 Gestione Oggetti

Come per la gestione degli utenti per far sì che la tabella di store comparisse nel sito di amministrazione di Django è stato necessario implementare nel file *admin.py*

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

admin.site.register(Item)
```

Figura 3.12: Admin Gestione Oggetti.

In questa versione non sono presenti differenze rispetto la versione precedente se non l'utilizzo della classe utente personalizzata invece della classe standard di Django

```
from gestione_utenti.models import CustomUser as
    User
```

3.2.3 Gestione Assistenza

Per aggiungere un ulteriore servizio al progetto, in modo da poter implementare in seguito un terzo microservizio, è stata pensata e ideata una sezione per le domande da parte degli utenti.

Nel file HTML di base è stato aggiunto

```
<a class="assistenza" href="{% url 'send_message' %}">Serve assistenza?</a>
```

In questo modo in ogni pagina dell'applicazione comparirà

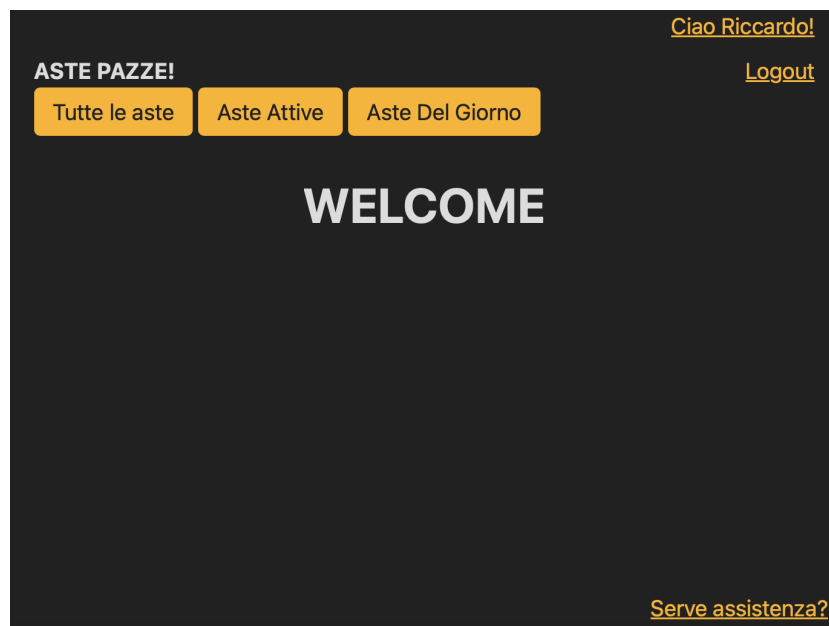


Figura 3.13: Link assistenza.

Cliccando sulla scritta ” *Serve assistenza?*” l’utente verrà reindirizzato in una pagina dove sarà possibile mandare un messaggio allo staff del sito, una volta mandato il messaggio se l’operazione va a buon fine comparirà una scritta informativa

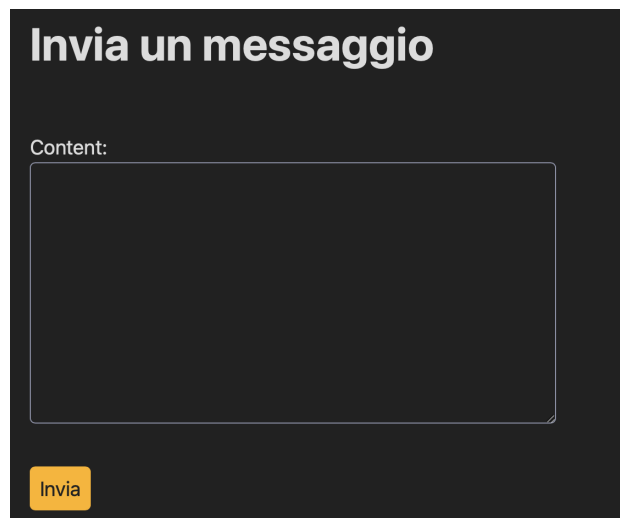
The image shows a dark-themed web form titled "Invia un messaggio" in white bold text. Below the title, the label "Content:" is followed by a large, empty rectangular text input area with a thin white border. At the bottom left of the form, there is a small orange button with the word "Invia" in white text.

Figura 3.14: Invio messaggio assistenza.

The image shows the same dark-themed web form as Figure 3.14, but with a success message displayed below the "Invia" button. The message is a bulleted list item: "• Il messaggio è stato inviato con successo! Ti risponderemo al più presto controlla l'email utilizzata nella fase di registrazione." The "Invia" button remains visible above the message.

Figura 3.15: Messaggio invio andato a buon fine.

La classe dei messaggi è la seguente

```
class Message(models.Model):
    sender = models.ForeignKey(User, related_name='sender', on_delete=models.CASCADE)
    content = models.TextField()
    timestamp = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return f"Sender:{self.sender} -> {self.content}"
```

Figura 3.16: Classe Message.

Il codice che gestisce i messaggi è implementato all'interno del file *view.py*

```
@login_required
def send_message(request):
    if request.method == 'POST':
        form = MessageForm(request.POST)
        if form.is_valid():
            message = form.save(commit=False)
            message.sender = request.user
            message.save()
            messages.success(request, "Il messaggio
                e' stato inviato con successo! Ti
                risponderemo al piu presto controlla
                l'email utilizzata nella fase di
                registrazione.")
            return redirect('send_message')
    else:
        form = MessageForm()
    return render(request, 'send_message.html', {'
        form': form})
```

Come si può vedere tramite *@login_required* solo gli utenti che hanno effettuato il login potranno effettuare delle domande.

Come per le altre applicazioni è stato necessario inserire all'interno del file *admin.py* per la visualizzazione della tabella

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

admin.site.register(Message)
```

Come mostrato nella figura 3.10 i membri dello staff possono accedere ad un link dove visualizzano i messaggi. Per rendere più sicuro l'accesso a questa pagina il link compare solo se l'utente fa parte dello staff, controllo effettuato nel file HTML relativo

```
{% if user.is_staff %}
    <p>Fai parte dello staff.</p>
    <a href="{% url 'message_list' %}"> Casella
        messaggi </a>
{% endif %}
```

Un ulteriore controllo viene effettuato nella vista

```
def message_list(request):
    if request.user.is_staff:
        messages = Message.objects.all()
        return render(request, 'message_list.html', context={'messages': messages})
    else:
        return render(request, 'home.html')
```

Figura 3.17: Vista message.

Capitolo 4

Applicazione con microservizi

In quest ocapitoo verrà mostrata l'applicazione implementata tramite la filosofia dei microservizi. Come mostrato nei capitoli precedenti le parti dell'applicazione sono tre:

- Gestione utenti;
- Gestione oggetti;
- Gestione assistenza.

Per ognuno di essi è stato creato un progetto Django univoco. Si è optato per eseguire le applicazioni in localhost cambiando la porta di esecuzione. Come applicazione principale è stata scelta quella per la gestione degli utenti, la quale applicazione avrà anche il compito di generare token e di consumare le richieste *API*.

Le tre applicazioni vengono eseguite ai seguenti indirizzi http, l'ordine segue quello precedente:

- *http://127.0.0.1:8000/*;
- *http://127.0.0.1:8001/*;
- *http://127.0.0.1:8002/*.

L'applicazione per gli oggetti espone due url per la gestione delle richieste API, mentre la gestione assistenza espone solo un indirizzo.

Per far funzionare l'applicazione con i microservizi è stato necessario scaricare i seguenti pacchetti: *rest_framework* [2], *corsheaders* [3] e *jwt* [4].

Per poter decodificare il token in tutte le applicazioni è stata impostata la **stessa chiave segreta**.

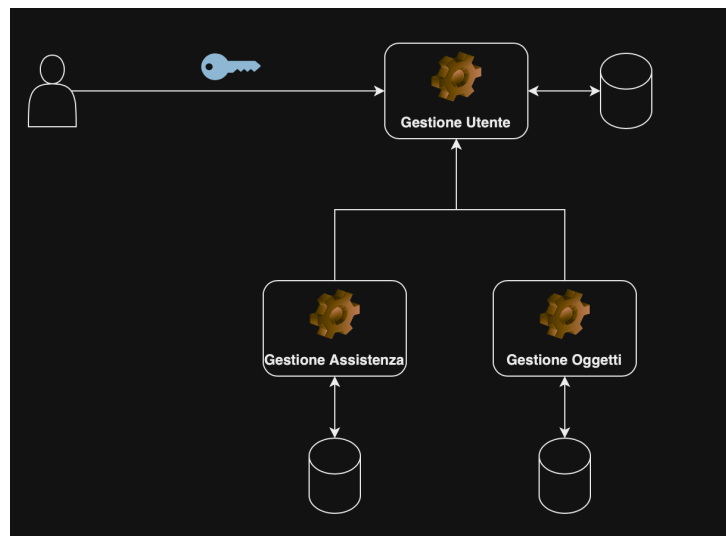


Figura 4.1: Schema Architettuale.

In tutte e tre le applicazioni sono state aggiunte le seguenti applicazioni all'interno di **INSTALLED_APP** nel file *settings.py*

```

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'gestione_utenti.apps.GestioneUtentiConfig',
    'rest_framework',
    'corsheaders',
]
  
```

Figura 4.2: Installed app.

Per permettere le CORS tra le applicazione bisogna aggiungere i seguenti campi

Per far sì che funzioni tutto il *corsheaders.middleware.CorsMiddleware* dev'essere inserito prima di *django.contrib.messages.middleware.MessageMiddleware*. Nelle sezioni successive verranno mostrate le modifiche riportate, il resto sarà come nella seconda versione.


```
MIDDLEWARE = [  
    'corsheaders.middleware.CorsMiddleware',  
    'django.middleware.security.SecurityMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.csrf.CsrfViewMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
    'django.contrib.messages.middleware.MessageMiddleware',  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
]  
  
CORS_ALLOW_ALL_ORIGINS = True
```

Figura 4.3: Abilitazione CORS.

4.1 Gestione Utenti

Nell'applicazione per la gestione utenti è stata definita la seguente view:

```
@csrf_exempt
def view(request):
    if 'Authorization' in request.headers:
        auth_header = request.headers['Authorization']
        _, token = auth_header.split(' ')

        try:
            decoded_token = jwt.decode(token,
                                         settings.SECRET_KEY, algorithms=['HS256'])
            user_email = decoded_token.get('user_email')
            username = decoded_token.get('user_username')

            message = f"Richiesta ricevuta con successo. Utente associato all'email: {user_email}, con username: {username}"

            user = User.objects.get(email=user_email)
            login(request, user)

            return JsonResponse({"message": message})
        except jwt.ExpiredSignatureError:
            return JsonResponse({"error": "Token scaduto"}, status=401)
        except jwt.InvalidTokenError:
            return JsonResponse({"error": "Token non valido"}, status=401)
    else:
        return JsonResponse({"error": "Token mancante nell'header Authorization"}, status=401)
```

Il decorator `@csrf_exempt` indica che una vista è esente dalla verifica del token CSRF (Cross-Site Request Forgery).

La protezione CSRF è un meccanismo di sicurezza che prevenire attacchi in cui un malintenzionato tenta di far eseguire azioni non autorizzate da parte di un utente autenticato. Quando un utente autenticato fa una richiesta al server, Django include un token CSRF nella richiesta, e questo token deve corrispondere a quello memorizzato lato server. L'uso di `@csrf_exempt` indica che la vista decorata non deve essere soggetta a questa verifica del token CSRF. La seguente vista viene esposta in modo da poter essere chiamata dai vari microservizi.

4.1.1 Generazione Token

Il token che viene utilizzato per identificare gli utenti negli altri microservizi viene generato tramite la seguente funzione

```
def generate_jwt_token(user):  
    return jwt.encode({'user_email': user.email, '  
                        user_username': user.username}, SECRET_KEY,  
                      algorithm='HS256')
```

Questa funzione inserisce nel token l'informazione inerente all'*email* e l'*username* dell'utente.

Il token viene generato durante la fase di login, e salvato nella classe User

```
class LoginView(auth_views.LoginView):  
    form_class = LoginForm  
    template_name = 'Profilo/login.html'  
  
    def form_valid(self, form):  
        user = form.get_user()  
        login(self.request, user)  
  
        token = generate_jwt_token(user)  
        user.set_token(token)  
        response = super().form_valid(form)  
        response.data = {'token': token}  
        user.save()  
  
        return redirect('home')
```

In questo modo dopo ogni login il token sarà aggiornato, oppure durante la fase di registrazione

```
def register_view(request):  
    if request.method == 'POST':  
        form = RegisterForm(request.POST)  
        if form.is_valid():  
            user = form.save()  
  
            login(request, user)  
            token = generate_jwt_token(user)  
            user.set_tk(token)  
            user.save()  
            return redirect('home')  
    else:  
        form = RegisterForm()  
  
    return render(request, 'Profilo/register.html',  
                  {'form': form})
```

Per l'importanza del token nella comunicazione tra i microservizi il funzionamento delle funzionalità dell'applicazione sono permesse solo se si è effettuato l'accesso.

4.1.2 Chiamate Ajax

La comunicazione tra microservizi è stata gestita tramite le chiamate **ajax**, viene riportato un esempio di una sola delle chiamate.

```
$(document).ready(function() {  
    function getTokenFromUser() {  
        return '{{ request.user.tk }}';  
    }  
  
    function inviaTokenAlMicroservizio(token) {  
        $.ajax({  
            url: 'http://127.0.0.1:8001/tua-view/',  
            method: 'GET',  
            headers: {  
                'Authorization': 'Bearer ' + token  
            },  
            success: function(response) {  
                console.log('Token inviato con successo:',  
                    response);  
                window.location.href = "http  
                    ://127.0.0.1:8001/tutte_aste/"  
            },  
            error: function(xhr, status, error) {  
                console.log(header);  
                console.error('Errore durante l\'invio  
                    del token:', error);  
                console.log('Stato HTTP:', xhr.status);  
            }  
        });  
    }  
  
    $('#tutteAsteButton').on('click', function() {  
        const token = getTokenFromUser();  
        inviaTokenAlMicroservizio(token);  
    });  
});
```

4.2 Gestione Oggetti

4.3 Classe Quest

Per gestire l'utente all'interno del microservizio tramite l'usilio del token è stata definita una classe chiamata **Quest**

```
class Quest(models.Model):
    tk = models.CharField(max_length=100000)

    def set_tk(self, new_tk):
        self.tk = new_tk

    def get_tk(self):
        return self.tk

    def get_username(self):
        decoded_token = jwt.decode(self.tk, settings
            .SECRET_KEY, algorithms=['HS256'])
        return decoded_token.get('user_username')

    def get_email(self):
        decoded_token = jwt.decode(self.tk, settings
            .SECRET_KEY, algorithms=['HS256'])
        return decoded_token.get('user_email')

    def get_name(self):
        decoded_token = jwt.decode(self.tk, settings
            .SECRET_KEY, algorithms=['HS256'])
        return decoded_token.get('user_name')
```

Questa classe ha come attributo il solo token relativo all'utente e come funzioni delle classi di supporto atte alla decodifica del token per poter risalire alle informazioni relative all'utente.

4.3.1 Definizione View

Nell'applicazione per la gestione degli oggetti sono state implementate due view, la prima utilizzata per la comunicazione e per l'esposizione degli item disponibili e la seconda atta al ritornare gli oggetti posseduti o vinti dall'utente specifico, queste view sono esposte a degli specifici URL.

```
@csrf_exempt
def tua_view(request):
    if 'Authorization' in request.headers:
        auth_header = request.headers['Authorization']
        _, token = auth_header.split(' ')

        try:

            decoded_token = jwt.decode(token,
                settings.SECRET_KEY, algorithms=['HS256'])
            user_email = decoded_token.get('user_email')
            username = decoded_token.get('user_username')
            name = decoded_token.get('user_name')

            quest = Quest()
            quest.set_tk(token)
            quest.save()

            message = f"Richiesta ricevuta con
                successo. Utente associato all'email:
                {user_email}, con username: {
                username}"
            return JsonResponse({"message": message
                })
        except jwt.ExpiredSignatureError:
            return JsonResponse({"error": "Token
                scaduto"}, status=401)
        except jwt.InvalidTokenError:
            return JsonResponse({"error": "Token non
                valido"}, status=401)
    else:
```

```
return JsonResponse({"error": "Token  
mancante nell'header Authorization"},  
                    status=401)
```

```
@csrf_exempt  
def view_oggetti(request):  
    if 'Authorization' in request.headers:  
        auth_header = request.headers['Authorization']  
        _, token = auth_header.split(' ')  
  
    try:  
  
        decoded_token = jwt.decode(token,  
                                    settings.SECRET_KEY, algorithms=['  
HS256'])  
        user_email = decoded_token.get('user_email')  
        username = decoded_token.get('user_username')  
        name = decoded_token.get('user_name')  
  
        quest = Quest()  
        quest.set_tk(token)  
        quest.save()  
  
        object_list = Item.objects.all()  
        oggetti_aggiudicati = [item for item in  
                                object_list if item.check_User(quest.  
get_email()) == True and item.  
available() == False]  
        oggetti_aggiudicati = [{"text": item.  
text, "min_sale": item.min_sale, "  
offer": item.offer} for item in  
                                oggetti_aggiudicati]  
  
        return JsonResponse({"oggetti_aggiudicati": str(  
oggetti_aggiudicati)})  
    except jwt.ExpiredSignatureError:
```



```
        return JsonResponse({"error": "Token  
scaduto"}, status=401)  
    except jwt.InvalidTokenError:  
        return JsonResponse({"error": "Token non  
valido"}, status=401)  
    else:  
        return JsonResponse({"error": "Token  
mancante nell'header Authorization"},  
                             status=401)
```

Nella seconda view il codice

```
oggetti_aggiudicati = [{"text": item.text, "min_sale  
": item.min_sale, "offer": item.offer} for item  
in oggetti_aggiudicati]
```

Serve per la generazione di un json che sarà mandato all'applicazione *Gestione utenti* per far visualizzare gli oggetti vinti.

4.3.2 Chiamate Ajax

Come per la *Gestione Utenti* la comunicazione tra microservizi è stata gestita tramite le chiamate Ajax.

4.4 Gestione Assistenza

Nell'applicazione per la gestione dell'assistenza è stata utilizzata la classe **Quest** 4.3 per la gestione degli utenti tramite token ed è stata esposta una view per la comunicazione tra le differenti applicazioni.

Capitolo 5

UML

In questo capitolo sono mostrati gli schemi UML delle classi utilizzate nel progetto, gli UML sono inerenti all'ultima versione del progetto.

5.1 Gestione Utenti

5.1.1 CustomUser

Come discusso nel capitolo corrispondente vengono definite due classi CustomUserManager 3.2.1.1 e CustomUser 3.2.1.1.

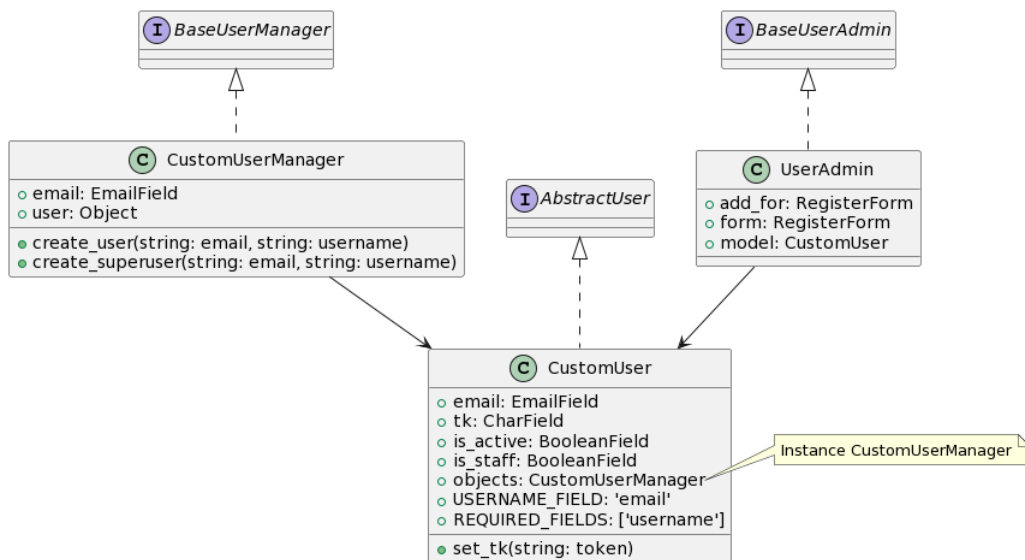


Figura 5.1: UML User.

Come mostrato nel formato UML per definire un superuser3.2.1.1 viene utilizzato il CustomUser e non l'user nativo di Django.

5.1.2 Registrazione e Login

Per la fase di login e registrazione come mostrato precedentemente vengono utilizzati i form e le view, nell'uml sottostante viene mostrato il collegamento tra le varie componenti.

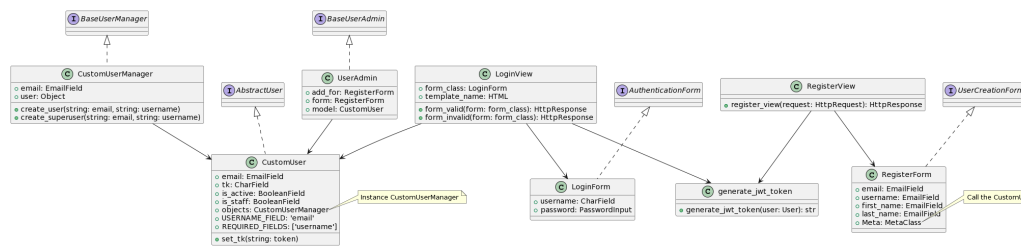


Figura 5.2: UML User.

Per vedere meglio il file UML con le componenti relative solo al login e register è stato definito il seguente UML

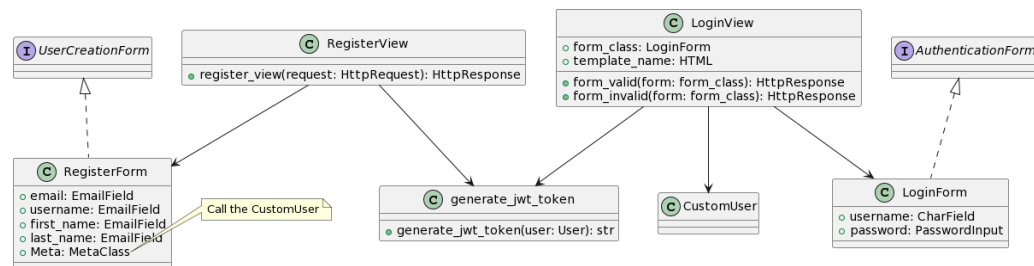


Figura 5.3: UML User.

5.2 Gestione Oggetti

5.2.1 Definizioni Classi

All'interno di questa applicazione vengono utilizzate due classi Item 2.21 e Quest 4.3

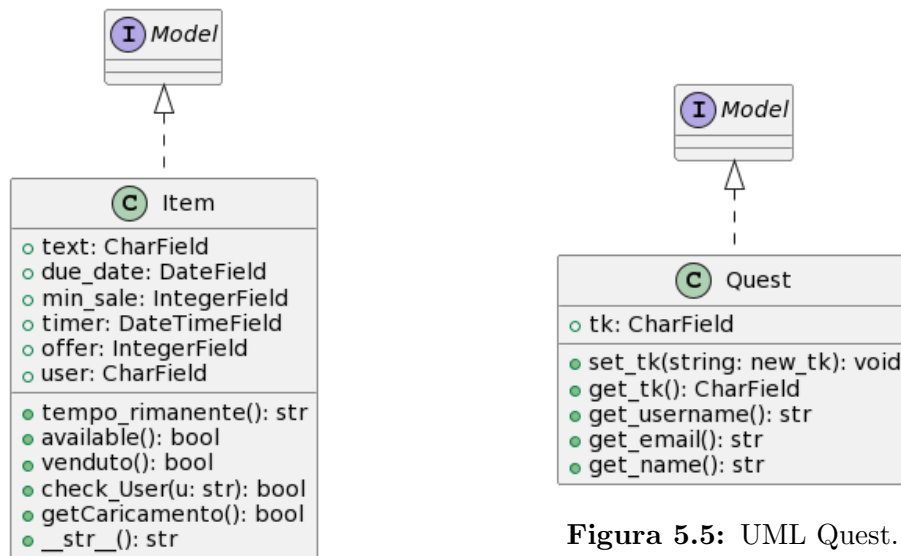


Figura 5.5: UML Quest.

Figura 5.4: UML Item.

5.2.2 View

Nell'UML seguente viene mostrato come comunicano tra loro le view per le rappresentazioni

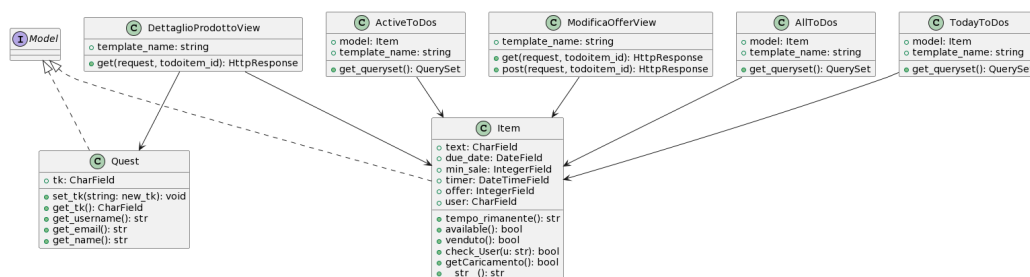


Figura 5.6: UML User.

5.3 Gestione Assistenza

5.3.1 Definizioni Classi

Le classi utilizzate nella seguente applicazione sono Message e Quest

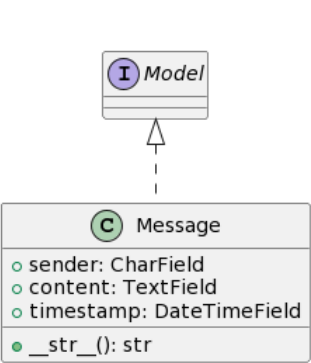


Figura 5.7: UML Message.

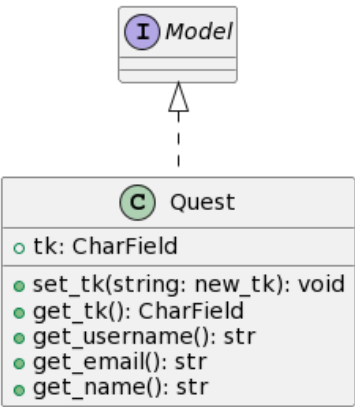


Figura 5.8: UML Quest.

5.3.2 View

Nel file UML seguente viene mostrata la comunicazione tra le view dell'applicazione gestione assistenza.

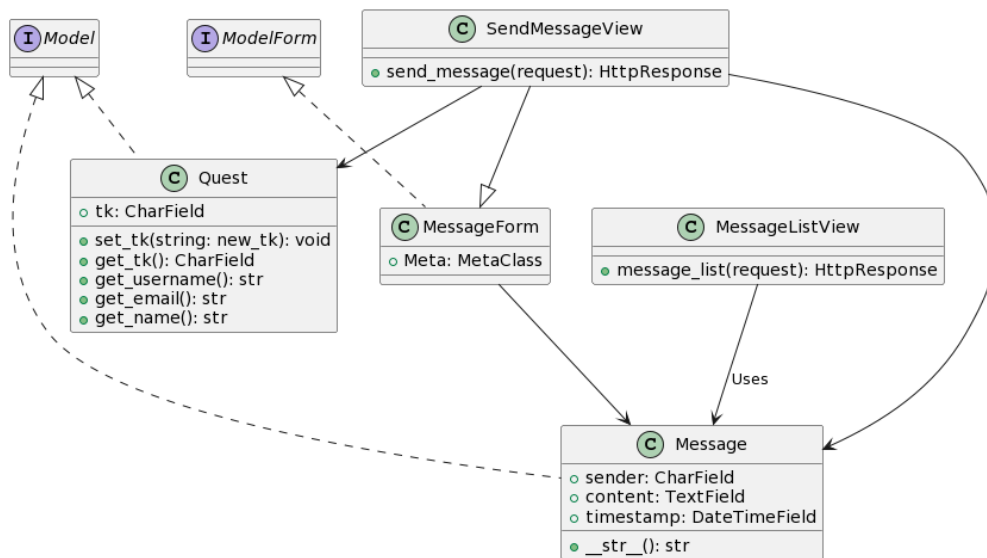


Figura 5.9: UML User.

Conclusione

Nel corso di questo progetto, sono state sviluppate e valutate tre differenti iterazioni, ciascuna rappresentante una prospettiva unica sull'architettura del sistema.

La prima iterazione si configura come una versione monolitica, caratterizzata dall'integrazione di tutte le funzionalità all'interno di un singolo progetto. In questa fase, il database unico non distingue gli elementi in base alle diverse funzionalità, adottando un approccio centralizzato.

La seconda iterazione, invece, adotta una strategia più raffinata, capitalizzando appieno sulle capacità di Django. Qui, le responsabilità sono distribuite tra diverse applicazioni all'interno del progetto. Ciascuna applicazione è responsabile di un ambito specifico, e il database è partizionato in modo coerente. Nonostante questo miglioramento, l'implementazione non raggiunge ancora il livello di una vera architettura a microservizi.

La terza iterazione, infine, abbraccia completamente la logica a microservizi. Tre distinti progetti Django sono stati creati, ciascuno dedicato alla realizzazione di un'applicazione autonoma. La comunicazione tra queste applicazioni avviene attraverso tecnologie avanzate come REST API e chiamate Ajax, permettendo una divisione chiara delle funzionalità in microservizi indipendenti.

Questa progressione rappresenta un approccio riflessivo nell'evoluzione dell'architettura del sistema, passando da una soluzione monolitica iniziale a una distribuzione avanzata a microservizi, riflettendo l'attenzione agli aspetti di modularità, scalabilità e separazione delle responsabilità nell'implementazione.

Bibliografia

- [1] Adrian Holovaty and Jacob Kaplan-Moss. *The Django Book*. Apress, 2009.
- [2] Django REST framework. Django rest framework documentation, Anno.
- [3] Adam Johnson. django-cors-headers documentation, Anno.
- [4] José Padilla. python-jose documentation, Anno.