



Django Users

Auth, Groups & permessi tra gli utenti



Tramite Form, FBV e CBV gli utenti possono...

aggiornare le tabelle dei DB della nostra web app in tutta comodità

Ma chi è l'utente? E chi lo gestisce?

Siamo sicuri di voler lasciare ad un generico “utente” questa libertà?

Per questi motivi, django ci mette a disposizione gli strumenti *auth*.

In settings.py

Tra le app e middleware installate “gratis” e di default nel momento in cui si crea un nuovo progetto Django:

```
INSTALLED_APPS = [
```

```
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'crispy_forms',  
    'gestione'
```

```
]
```

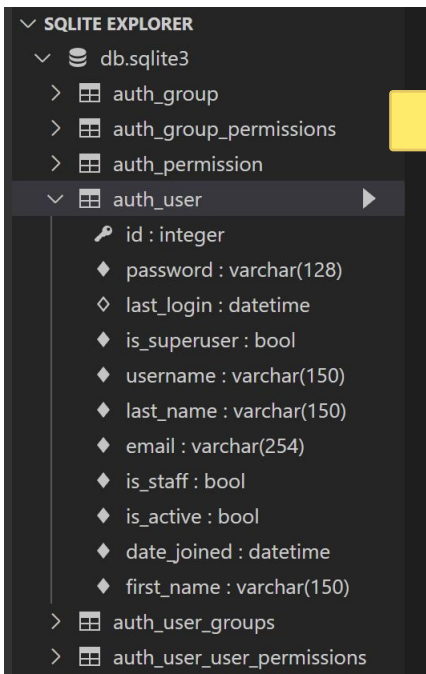
```
MIDDLEWARE = [
```

```
    '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',
```

```
]
```

Cosa ne consegue?

La creazione della tabella “User” all’interno del file db che si crea\popola dopo il primo comando di migrazione (*python manage.py migrate*)



Quanti utenti abbiamo?

Quanti utenti? Uno solo

creato con *python manage.py createsuperuser*, ovverosia l'**admin**.

SQL ▾								
< 1 / 1 > 1 - 1 of 1								
id	password	last_login	is_superuser	username	last_name	email	is_staff	is_active
1	pbkdf2_sha256\$320000\$vI32Ve3UcpDf8H6b0jpAva\$xfQr8V9LrTEK0agx4IBHKJf6UrBhCwxkdXEVUWc63ZE=	2022-02-16 10:31:51.217602	1	admin		admin@admin.it	1	1



“Accedere” agli utenti

```
def function_view(request, ...):
```

```
    request.user.username/etc . . .
```

```
class CBView([Create/List/Update/Delete]View):
```

```
    self.request.user . . . #all'interno di un metodo
```

```
# da template:
```

```
user.<attrib>
```

request.user

Ci restituisce “Anonymous” nel caso l’utente non sia loggato.

Ci restituisce “admin” nel caso in cui, dopo aver creato il superutente admin, andiamo su *localhost:8000/admin/* ed eseguiamo il login da lì.

Come il superuser, vorremmo poter creare utenti. Questa volta però voglio avere utenti “normali”, **ossia senza privilegi di staff ed accesso al pannello di controllo admin.**

Come si crea un User?

- In caso l'utente sia parte dello staff, può creare un User:
 - Tramite pannello di controllo admin
- In caso l'utente sia anonymous, può registrarsi alla webapp
 - Tramite FBV & CBV (e.g. CreateView...)



Questo non ci deve stupire. Similmente a come abbiamo trattato tutte le entity che abbiamo visto nelle lezioni precedenti (libri, studenti, domande, risposte etc...)

Del resto un “User” non è altro che una entry di una table di un model. E noi sappiamo come si gestiscono...

L'importanza di User

La tabella User è di fondamentale importanza per una webapp.

Ci consente di subordinare l'accesso alle risorse (dati all'interno dei DB) della nostra webapp a logiche legate ai **permessi**.

Tipicamente:



Non si consente ad un utente non registrato di fare modifiche al database.

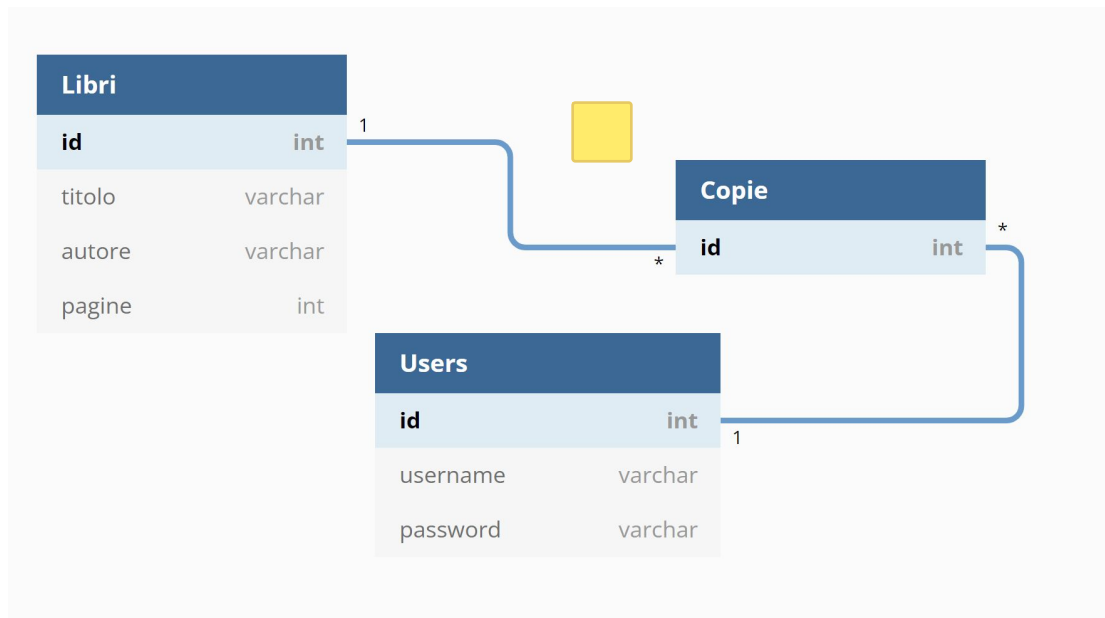
Esistono utenti di categorie diverse...

La gestione degli users è il primo passo per poter implementare una gestione di permessi

Esempio complesso (...django/biblio3auth/...)

Faremo una terza versione, l'ultima, della gestione della biblioteca. Questa volta con gestione degli utenti

Quindi creazione progetto biblio3, con tanto di app gestione. Come cambiano i models?



In altre parole

La biblioteca è composta da un certo numero di **libri**. Ciascun libro è associato a N **copie**. Ciascuna copia può essere attualmente in prestito o oppure no. Il prestito è operato da un **utente registrato**.

Rispetto a biblio2: non ci sono cambiamenti alla tabella Libro.

Ci sono due cambiamenti nella tabella Copia:

1. Togliamo la logica del prestito scaduto. Rimane solo la data dell'ultimo prestito
2. Aggiungiamo un attributo a Copia; sarà legato tramite foreignKey alla tabella **User**

Un progetto completo

Al netto della complessità, “facciamo finta” di ragionare in ottica di progettone finale da portare all’esame.

Quindi oltre ai requisiti della nostra applicazione, pensiamo anche a fare qualcosa di **facilmente navigabile**.

Come bonus: usiamo i crispy forms & bootstrap per cercare con uno sforzo minimo di presentare un frontend più accattivante rispetto a quello che abbiamo visto fino ad ora.

Funzionalità e specifiche

1) Chiunque può accedere alla biblioteca

- a) Può vedere quali libri ci sono e sapere se sono disponibili al prestito oppure no
- b) Può fare una ricerca per titolo o autore di un libro
- c) Può registrarsi come lettore

2) Il lettore può interagire con le copie

- a) Prestito
- b) Restituzione
- c) Situazione prestiti

3) Il Bibliotecario può interagire con i libri e le copie

- a) Aggiunta di un libro alla biblioteca
- b) Aggiunta di una copia di un libro alla biblioteca
- c) E' in grado di vedere la situazione della biblioteca: chi ha in prestito cosa
- d) Prestito/restituzione



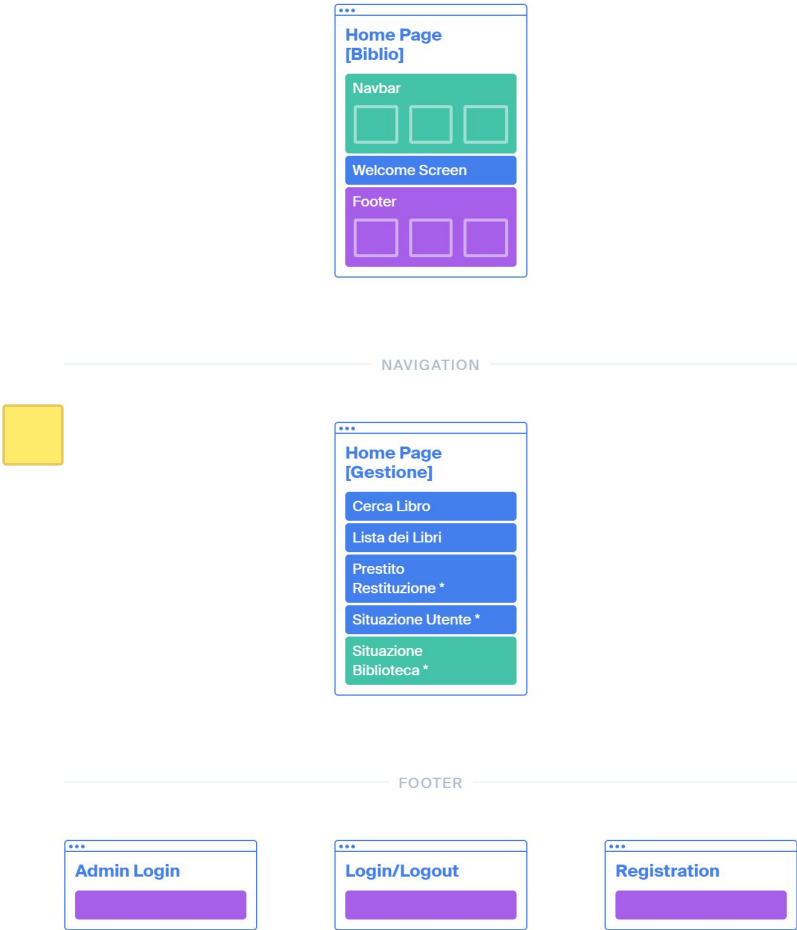
4) L'Admin

- a) Registra i bibliotecari
- b) Ha pieno controllo su tutto

Navigabilità

Con l'asterisco sono segnate le funzionalità che presumono la registrazione dell'utente.

In BLU, le sezioni raggiungibili dai lettori.
In Verde le sezioni raggiungibili dai bibliotecari



Operazioni iniziali

Prima di “risolvere” ciascun punto delle funzionalità che abbiamo elencato, occorre ovviamente fare il solito preambolo a cui siamo abituati e che, almeno in queste slides vi risparmio.

Partiamo subito con i modelli

Libro in gestione/models.py

```
class Libro(models.Model):
    titolo = models.CharField(max_length=200)
    autore = models.CharField(max_length=50)
    pagine = models.IntegerField(default=100)

    def disponibile(self):
        if self.copie.filter(data_prestito=None).count() > 0:
            return True
        return False

    def __str__(self):
        disp = self.copie.filter(data_prestito=None).count()
        out = self.titolo + " di " + self.autore + " ha " + str(self.copie.all().count()) +
            " copie" + " di cui " + str(dispatch) + " disponibili"
        return out

class Meta:
    verbose_name_plural = "Libri"
```


Copia in gestione/models.py

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

class Copia(models.Model):
    data_prestito = models.DateField(default=None, null=True, blank=True)
    libro = models.ForeignKey(Libro, on_delete=models.CASCADE, related_name="copie")
    utente = models.ForeignKey(User,
                                on_delete=models.PROTECT, blank=True, null=True, default=None,
                                related_name="copie_in_prestito")

    def chi_in_prestito(self):
        if self.utente == None: return None
        return self.utente.username

    def __str__(self):
        return "Copia di " + self.libro.titolo + " di " + self.libro.autore + " in prestito dal "
            + str(self.data_prestito)

    class Meta:
        verbose_name_plural = "Copie"
```

Possiamo cominciare...

... con quello che sappiamo fare:

- 1) Un comando iniziale di popolamento del database libri/copie.
- 2) Home di biblio: Con Navbar e footer per raggiungere facilmente le nostre funzionalità
- 3) Home di gestione: Per poter permettere all'utente di interagire con le copie.
 - a) Dovremo in realtà prima implementare la registrazione dell'utente per prestiti\restituzioni,
 - b) ma possiamo comunque implementare per intero la ricerca e il dump del database dei libri

Home di Biblio3

Pagina iniziale della biblioteca. Ora con gestione utenti!

Biblio3 Gestione Lista dei Libri Cerca un Libro

Homepage Biblio

Versione 3: con CBV, Frames e Auth



Comincia la navigazione

Registrati

Login

Admin Login

HEADER

NAVBAR



WELCOME SCREEN

FOOTER

Home di Gestione

Welcome Utente non registrato

Biblio3

Gestione

Lista dei Libri

Cerca un Libro

Cosa vuoi fare oggi?

Cerca un libro

Lista dei titoli

La mia situazione

Registrati

Login

Admin Login

HEADER

NAVBAR

CONTENT

FOOTER

Lista dei Libri

Cerca un Libro

La nostra biblioteca possiede 6 libri

Libro attualmente non disponibile

Promessi Sposi

Un libro di Alessandro Manzoni di ben 832 pagine...

Prendilo in Prestito!

Libro disponibile per il prestito

1984

Un libro di George Orwell di ben 328 pagine...

Prendilo in Prestito!

Libro disponibile per il prestito

Odissea



In gestione/views.py

```
def gestione_home(request):  
    return render(request,template_name="gestione/home.html")  
  
class LibroListView(ListView):  
    titolo = "La nostra biblioteca possiede"  
    model = Libro  
    template_name = "gestione/lista_libri.html"
```

In templates/gestione/libri.html

```
[...]
{% for l in object_list %}
<br>
<div class="card">
  <div class="card-header">
    {% if l.disponibile %}
      Libro disponibile per il prestito
    {% else %}
      Libro attualmente non disponibile
    {% endif %}
  </div>
  <div class="card-body">
    <h5 class="card-title">{{ l.titolo }}</h5>
    <p class="card-text">Un libro di {{ l.autore }} di ben {{ l.pagine }} pagine...</p>
    {% if l.disponibile %}
      <a href="{% url 'gestione:prestito' l.pk %}" class="btn btn-info">
        {% else %}
      <a href="#" class="btn btn-info disabled">
    {% endif %}
    Prendilo in Prestito!</a>
  </div>
</div>
<br>
{%endfor%}
[...]
```

https://www.w3schools.com/bootstrap4/bootstrap_cards.asp

SearchForm, in gestione/forms.py

Cerca in Titolo o Autore di un Libro

Nessuna differenza sostanziale con quanto abbiamo visto nelle slides inerenti i Django Form. Useremo l'Helper datoci in regalo da django-crispy-forms. **Si ricorda di importare i moduli appositi (from crispy_forms.helper import FormHelper & from crispy_forms.layout import Submit)**

```
class SearchForm(forms.Form):

    CHOICE_LIST = [("Titolo","Cerca tra i titoli"), ("Autore","Cerca tra gli autori")]
    helper = FormHelper()
    helper.form_id = "search_crispy_form"
    helper.form_method = "POST"
    helper.add_input(Submit("submit","Cerca"))
    search_string = forms.CharField(label="Cerca qualcosa",max_length=100,
                                    min_length=3, required=True)
    search_where = forms.ChoiceField(label="Dove?", required=True, choices=CHOICE_LIST)
```


In gestione/views.py

```
def search(request):

    if request.method == "POST":
        form = SearchForm(request.POST)
        if form.is_valid():
            sstring = form.cleaned_data.get("search_string")
            where = form.cleaned_data.get("search_where")
            return redirect("gestione:ricerca_risultati", sstring, where)
        else:
            form = SearchForm()

    return render(request, template_name="gestione/ricerca.html", context={"form": form})


class LibroRicercaView(LibroListView):
    titolo = "La tua ricerca ha dato come risultato"

    def get_queryset(self):
        sstring = self.request.resolver_match.kwargs["sstring"]
        where = self.request.resolver_match.kwargs["where"]

        if "Titolo" in where:
            qq = self.model.objects.filter(titolo__icontains=sstring)
        else:
            qq = self.model.objects.filter-autore__icontains=sstring)

    return qq
```

templates/gestione/ricerca.html

```
{% extends 'basebs.html' %}

{% block title %} Cerca tra i libri{% endblock %}

{% load crispy_forms_tags %}
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css"
integrity="sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8ERdknLPMO"
crossorigin="anonymous">

{% block header1 %}
  <h1>Cerca tra i libri</h1>
{% endblock %}

{% block content %}

<h2> Cerca tra i libri </h2>

<br>
<div class="d-flex justify-content-center">

  {% crispy form %}

</div>
<br>
{% endblock %}
```

Cerca tra i libri

Cerca tra i libri

Cerca qualcosa*

Dove?*

Cerca tra gli autori

Cerca

Registrati

Login

Admin Login

Lista dei Libri

Libro

La tua ricerca ha dato come risultato 2 libri

Libro disponibile per il prestito

Odissea

Un libro di Omero di ben 414 pagine...

Prendilo in Prestito!

Libro disponibile per il prestito

Iliade

Un libro di Omero di ben 263 pagine...

Prendilo in Prestito!

Registrati

Login

Admin Login

Funzionalità e specifiche

- 1) Chiunque può accedere alla biblioteca
 - a) Può vedere quali libri ci sono e sapere se sono disponibili al prestito oppure no
 - b) Può fare una ricerca per titolo o autore di un libro
 - c) **Può registrarsi come lettore**

Registrazione utente

In django, di default, non solo possiamo sfruttare la tabella User, ma anche un sistema di registrazione/login/logout **quasi già completamente** implementato.

Questo sistema pre-esistente presume che l'utente si possa registrare sulla tabella User, la quale è accessibile a tutte le app del progetto. Occorrerà poi a noi sviluppatori inserire vincoli di permessi aggiuntivi.

Andremo quindi ad operare su `biblio3/urls.py`

```
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('admin/', admin.site.urls),
    re_path(r"^$|^\/$|^home\/$", biblio3_home, name="home"),
    path("gestione/", include("gestione.urls")),
    path("register/", UserCreateView.as_view(), name="register"),
    path("login/", auth_views.LoginView.as_view(), name="login"),
    path("logout/", auth_views.LogoutView.as_view(), name="logout"),
    ...
```

Registrazione in views.py (biblio3)

```
from django.contrib.auth.forms import UserCreationForm
from django.views.generic.edit import CreateView
from django.shortcuts import render
from django.urls import reverse_lazy

def biblio3_home(request):
    return render(request, template_name="home.html")

class UserCreateView(CreateView):
    form_class = UserCreationForm
    template_name = "user_create.html"
    success_url = reverse_lazy("login")
```

Come anticipato, è una createview che aggiunge un'entry alla tabella User. Useremo un ModelForm particolare che è dato "gratis" da django chiamato UserCreationForm. Appare come un ModelForm in cui **Meta.model = User**

in templates/user_create.html

```
...
{% block content %}

<form method="post"> {% csrf_token %}

    {{form | crispy}}

    <input type="submit" class="btn btn-success" value="Registrati ora!">

</form>

{% endblock %}
...
```

Lo UserCreateForm di django non usa il crispy Helper, quindi dobbiamo “crispizzare” manualmente noi il form. Non solo: dobbiamo inserire l’elemento input/submit e chiudere il form

Pagina di Registrazione

Si prega di compilare i campi necessari

Biblio3 Gestione Lista dei Libri Cerca un Libro

Username*

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

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 ora!

Da notare i password validators come elencati in settings.py. Non essendo questo un utente “speciale” (staff/admin) **non vi è modo di ignorarli come abbiamo fatto con createsuperuser!**

Click del pulsante submit

Supponendo i campi riempiti lato client passino i test di validazione, il sistema aggiunge l'username e la password come una entry nella tabella User. Di default, però, non porta a termine la procedura di login: infatti, quello che succede è che in seguito alla registrazione **l'utente viene redirezionato ad un URL che deve essere specificato nella variabile success_url.** 

```
success_url = reverse_lazy("login")
```

```
#ricorda che in biblio3/urls.py: path("login/", auth_views.LoginView.as_view(),  
                                     name="login"),
```

La view di login

E' una view la cui logica è completamente costruita e data in regalo da Django.

Difatti, abbiamo dovuto importare il modulo *django.contrib.auth*.

Essendo la **logica** già implementata in tale modulo, **non dobbiamo aggiungere nulla al nostro views.py**.

Dobbiamo però fornire noi il template.

Il template di login

Banalmente è un form da renderare in HTML. Lo raggiungiamo tramite la variabile di contesto aggiunta da django chiamata “form”. In altre parole, il template che scriveremo per la schermata di login è praticamente identico a quello già visto in `user_create.html`.

Problemi?

Ma io non ho scritto la view!


Non ho modo di impostare *quale* template e *dove* si trova.

Non ho modo di impostare un *success_url*.

Soluzione:

Non ne ho bisogno, perchè seguo le regole imposte dall'app *auth*

Il template deve chiamarsi *login.html* e **deve trovarsi in *biblio3/templates/registration***.

Il *success_url* deve essere hardcodato in una variabile che chiameremo ***LOGIN_REDIRECT_URL*** in *settings.py* 

In settings.py

```
LOGIN_REDIRECT_URL = "/" #redireziona alla home
```

Nel nostro caso, possiamo aggiungere una particolarità

```
LOGIN_REDIRECT_URL = "/?login=ok" #redireziona alla home, ma con un parametro GET
```

Questo ci consente di distinguere quando l'utente approda alla homepage da utente registrato oppure come anonymous.

/?login=ok nel template della home

```
{% if user.is_authenticated and "ok" in request.GET.login %}  
  
<div id="modal" class="modal fade" tabindex="-1" role="dialog">  
  <div class="modal-dialog" role="document">  
    <div class="modal-content">  
      <div class="modal-header">  
        <h5 class="modal-title">Welcome {{user.username}} </h5>  
        <button type="button" class="close" data-dismiss="modal" aria-label="Close">  
          <span aria-hidden="true">&times;</span>  
        </button>  
      </div>  
      <div class="modal-body">  
        <p> La procedura di login è andata a buon fine </p>  
      </div>  
      <div class="modal-footer">  
        <button type="button" class="btn btn-info" data-dismiss="modal">Chiudi</button>  
      </div>  
    </div>  
  </div>  
</div>  
  
<script>  
  $(document).ready(function(){  
    |   $("#modal").modal('show');  
  });  
</script>  
{% endif %}
```

Controllo autenticazione
utente e parametro GET in
template

bootstrap modal dialog:

https://www.w3schools.com/bootstrap4/bootstrap_modal.asp

JS: al caricamento della
pagina, mostra il dialog
con id=modal

Pagina di Login

Si prega di compilare i campi necessari

Username*

utente3

Password*

.....

Esegui login

Welcome utente3

×

La procedura di login è andata a buon fine

Chiudi

Homepage Biblio

Versione 3: con CBV, Frames e Auth

Logout

La view: anche in questo caso è completamente costruita e data in regalo da Django. Sempre dal modulo *django.contrib.auth*.

Anche in questo caso, la **logica** è già implementata in tale modulo, **non dobbiamo aggiungere nulla al nostro views.py**.

Il template lo daremo noi anche in questo caso. **Si DEVE chiamare logged_out.html sempre in templates/registration.**

Il template è un semplice messaggio di “Grazie ed arrivederci”. Non ha form o altri componenti interattivi da renderare. 

Pagina di Logout

Logout eseguito con successo


Ci vediamo presto!

Dove siamo?

- 1) Chiunque può accedere alla biblioteca
 - a) Può vedere quali libri ci sono e sapere se sono disponibili al prestito oppure no
 - b) Può fare una ricerca per titolo o autore di un libro
 - c) Può registrarsi come lettore
- 2) Il lettore può interagire con le copie
 - a) Prestito
 - b) Restituzione
 - c) Situazione prestiti

Le funzionalità del punto 2) presumono quindi che l'utente sia registrato e loggato. Non posso consentire ad un Anonymous di fare queste cose. Come lo risolvo?

Le funzionalità del punto 2: possibili soluzioni

1. Da template posso “nascondere” agli utenti non autenticati i link che portano alle view in grado di modificare le tabelle libro/copia. 
2. La mia view può di fatto accedere da *request* a *user*. E da *user* posso poi sapere se *is_authenticated* equivale a **True** o meno.

Le funzionalità del punto 2: possibili soluzioni

1. Da template posso “nascondere” agli utenti non autenticati i link che portano alle view in grado di modificare le tabelle libro/copia.

Soluzione non funzionante! Puoi anche nascondere il link sul template/HTML, ma se tali url sono “costruibili” con le regole descritte nei vari `urls.py`, **sono comunque raggiungibili**

2. La mia view può di fatto accedere da *request* a *user*. E da *user* posso poi sapere se *is_authenticated* equivale a **True** o meno.

Questa soluzione è funzionante, ma presuppone molto codice boiler-plate. In altre parole, se ho Nmila view da dover “proteggere” tramite login, devo accedere a `request.user` in ognuna di esse...

Soluzione ottimale: django auth decorators

Dovremmo poter fare un'operazione di pre-processing su ogni view da proteggere. In python esistono i decorator, ossia funzioni che prendono in ingresso delle funzioni e restituiscono una versione pre/post processata della funzione data in ingresso. Noi abbiamo bisogno di un pre-processing, e quest'ultimo ci è fornito da django.



in gestione/views.py

```
from django.contrib.auth.decorators import login_required
```

```
@login_required
```

```
def my_situation(request):  
    user = get_object_or_404(User, pk=request.user.pk)  
    copie = user.copie_in_prestito.all()  
    ctx = { "listacopie" : copie }  
    return render(request,"gestione/situation.html",ctx)
```

in localhost:8000/gestione/situation/

Home di utente2

Welcome utente2

Libro

Risulta che hai correntemente in prestito: 3 copie

Ulteriori dettagli

Copia con id 52
<div><div>Promessi Sposi</div><div>Un libro di Alessandro Manzoni di ben 832 pagine...</div><div>Restituiscimi</div></div>
Copia con id 54
<div><div>1984</div><div>Un libro di George Orwell di ben 328 pagine...</div><div>Restituiscimi</div></div>
Copia con id 55

Se l'utente è autenticato

E se non fosse loggato?

Viene redirezionato ad un url specificato in **settings.py**, nella variabile da aggiungere chiamata **LOGIN_URL**

```
LOGIN_URL = "/login/" #redireziona alla pagina di login
```

Nel nostro caso, possiamo aggiungere ancora una volta, una particolarità

```
LOGIN_URL = "/login/?auth=notok" #redireziona al login, ma con un parametro GET
```

Questo ci consente di distinguere quando l'utente approda alla pagina di login “per scelta” **oppure perchè ha tentato di accedere ad una pagina protetta dal decoratore @login_required.**

Pagina di Login

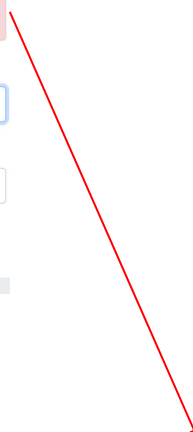
Si prega di compilare i campi necessari

Si è arrivati qui in quanto non si dispone dei permessi necessari per il servizio scelto!

Username*

Password*

Esegui login



```
{% if "notok" in request.GET.auth %}  
<div class="alert alert-danger" role="alert">  
| Si è arrivati qui in quanto non si dispone dei permessi necessari per il servizio scelto!  
</div>  
{% endif %}
```

in templates/registration/login.html

Esercizio

Implementare **una function view** che gestisca il prestito di una copia. Nell'app gestione.

- Tale view è raggiungibile tramite:
 - `path("prestito/<pk>/", prestito, name="prestito"), #gestione/urls.py`
- Ovviamente: **protetta dal decoratore di login**.
- Altri vincoli sono espressi nella slide successiva.
- Raggiungibile dopo una ricerca di un libro:

Libro disponibile per il prestito

1984

Un libro di George Orwell di ben 328 pagine...

Prendilo in Prestito!

Vincoli sull'esercizio

Il libro **deve** essere disponibile per il prestito. Ovverosia, deve avere almeno una copia associata **non attualmente in prestito**.

Un utente **non** può prendere in prestito **due copie dello stesso libro**. Con “stesso” s'intende, stesso titolo ed autore.

Ovviamente: la tabella/DB libro dovrà aggiornarsi in maniera consona.

Login required per le CBV

Le funzioni *prestito* e *my_situation* sono appunto **function views**. Come tali, sappiamo come agisce il meccanismo di python inerente il concetto di decoratori.

Ma come faccio a proteggere una **Class Based Views**? I decoratori si applicano a funzioni e per estensioni ai metodi. **Ma non ad intere classi**. E' comunque possibile usare i decoratori con le CBV, ma risulta scomodo e poco efficiente.

Per questo motivo, django mette a disposizione un altro meccanismo di protezione specifico per le CBV chiamato **access mixin**. Si basa sul concetto di ereditarietà multipla, la quale è supportata da python.

Vediamo ora come funziona questo meccanismo per il sistema di restituzione di una copia.

```
#in gestione/urls.py
path("restituzione/<pk>/", RestituisciView.as_view(), name="restituzione"),
```

```
#in gestione/views.py
from django.contrib.auth.mixins import LoginRequiredMixin

class RestituisciView(LoginRequiredMixin, DetailView):
    model = Copia
    template_name = "gestione/restituzione.html"
    errore = "NO_ERRORS"

    def get_context_data(self, **kwargs):
        ctx = super().get_context_data(**kwargs)
        c = ctx["object"]

        if c.data_prestito != None:

            if c.utente.pk != self.request.user.pk:
                self.errore = "Non puoi restituire un libro non tuo!"
            else:
                self.errore = "Libro attualmente non in prestito"

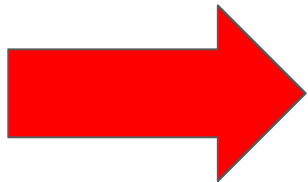
        if self.errore == "NO_ERRORS":
            try:
                c.data_prestito = None
                c.utente = None
                c.save()
            except Exception as e:
                print("Errore! " + str(e))
                self.errore = "Errore nell'operazione di restituzione"

        return ctx
```

LoginRequiredMixin

Aggiunge in automatico il decoratore `@login_required` ai metodi fondamentali ereditati da (in questo caso) *DetailView*.

Stessa logica vista per i decoratori nelle function views: “si passa” solo se si è autenticati. Altrimenti redirezione su LOGIN_URL come specificato su settings.py.



ATTENZIONE: l'ordine degli eredi della View che si sta costruendo **non è indifferente**. Gli access mixin devono essere **i primi** ad essere ereditati. Questo è dettagliato nella guida di django. Occhio a non fare a pugni con l'MRO (Method Resolution Order) di Python.

A che punto siamo?

1) Chiunque può accedere alla biblioteca

- a) Può vedere quali libri ci sono e sapere se sono disponibili al prestito oppure no
- b) Può fare una ricerca per titolo o autore di un libro
- c) Può registrarsi come lettore

2) Il lettore può interagire con le copie

- a) Prestito
- b) Restituzione
- c) Situazione prestiti



3) Il Bibliotecario può interagire con i libri e le copie

- a) Aggiunta di un libro alla biblioteca
- b) Aggiunta di una copia di un libro alla biblioteca
- c) E' in grado di vedere la situazione della biblioteca: chi ha in prestito cosa
- d) Prestito/restituzione

4) L'Admin

- a) Registra i bibliotecari
- b) Ha pieno controllo su tutto

Tipologie di Users

Dalle specifiche del nostro progetto si evince che abbiamo diverse tipologie di utenti.

- Registrati
 - Admin
 - Lettori
 - Bibliotecari



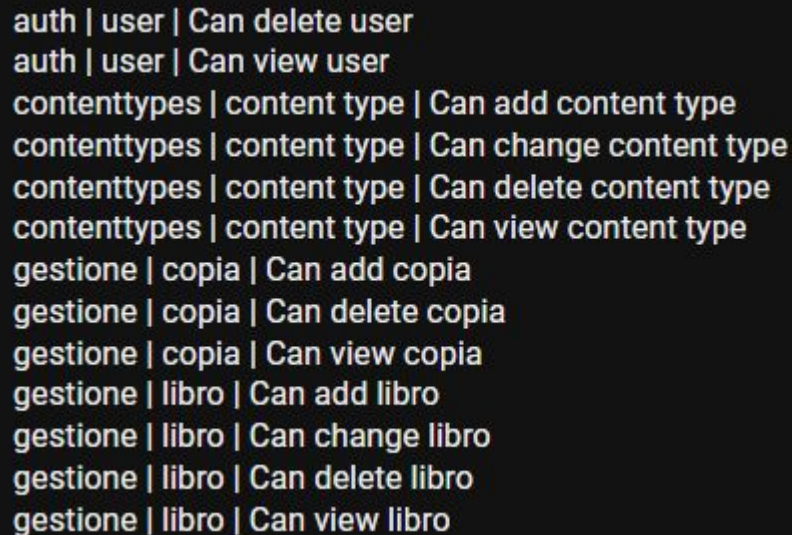
- Non Registrati

Come si traduce tutto questo in Django?

Django Groups & permissions

- Un **gruppo** in Django è un contenitore di **permessi**.
- Gli **Users** possono essere associati a nessuno, uno o a più gruppi, “ereditandone” i permessi
- I **permessi** appaiono come regole (o vincoli)
 - Possono essere assegnati agli **Users**
 - Possono essere assegnati ai **Gruppi**
 - Si applicano alle **risorse** della webapp (tabelle, DB, FBV/CBV)
 - Lettura, Scrittura, Rimozione, Aggiornamento etc...

La granularità dei permessi



```
auth | user | Can delete user
auth | user | Can view user
contenttypes | content type | Can add content type
contenttypes | content type | Can change content type
contenttypes | content type | Can delete content type
contenttypes | content type | Can view content type
gestione | copia | Can add copia
gestione | copia | Can delete copia
gestione | copia | Can view copia
gestione | libro | Can add libro
gestione | libro | Can change libro
gestione | libro | Can delete libro
gestione | libro | Can view libro
```

Dal pannello admin/superuser

Nella sua forma generica:

app | risorsa | azione

Possiamo (tramite codice) aumentarne la granularità a livello di attributo di tabella.

E' meccanismo tanto preciso quanto **poco scalabile**.

Per questo si usano i **gruppi**...



Creazione dei Gruppi

Programmaticamente da codice:

```
from django.contrib.auth.models import Group
gruppo_a = Group.objects.get_or_create(name='nome_gruppo')
```

#definizione dei permessi...

```
gruppo_a.permissions.set([permission_list])
gruppo_a.permissions.add(permission, permission, ...)
gruppo_a.permissions.remove(permission, permission, ...)
gruppo_a.permissions.clear()
```

#inserimento utente in gruppo...

```
group_a.user_set.add(user)
```

#oppure

```
user.groups.add(group_a)
```



Creazione dei Gruppi [Admin panel]

Django administration

Home > Authentication and Authorization > Groups

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

GESTIONE

Copie + Add

Libri + Add

Select group to change

Search

Action: Go 0 of 2 selected

GROUP

Bibliotecari

Lettori

2 groups

Lettori

Name: Lettori

Permissions:

Available permissions

Filter

admin | log entry | Can add log entry

admin | log entry | Can change log entry

admin | log entry | Can delete log entry

admin | log entry | Can view log entry

auth | group | Can add group

auth | group | Can change group

auth | group | Can delete group

auth | group | Can view group

auth | permission | Can add permission

auth | permission | Can change permission

auth | permission | Can delete permission

auth | permission | Can view permission

auth | user | Can add user

Chosen permissions

gestione | copia | Can change copia

Choose all

Remove all

Tornando al caso della biblioteca

Abbiamo quindi aggiunto i gruppi Lettori e Bibliotecari e assegnato utenti e permessi in maniera consona alle nostre specifiche.

Ha senso che un admin crei i gruppi. Lo farà solo durante lo sviluppo della webapp.

Ma non ha senso “spostare” manualmente gli Users nei gruppi opportuni. Sarebbe il caso di farlo in automatico con le nostre **createUsers** view.

Modifiche a biblio3/views.py -> UserCreateView

```
class UserCreateView(CreateView):  
    #form_class = UserCreationForm  
    form_class = CreaUtenteLettore  
    template_name = "user_create.html"  
    success_url = reverse_lazy("login")
```

In biblio3/forms.py



```
from django.contrib.auth.models import Group  
from django.contrib.auth.forms import UserCreationForm  
class CreaUtenteLettore(UserCreationForm):  
  
    def save(self, commit=True):  
        user = super().save(commit)  
        g = Group.objects.get(name="Lettori")  
        g.user_set.add(user)  
        return user
```

Idea: ereditiamo dal form “base” UserCreationForm

Facciamo un override del metodo **save** per assicurarci di assegnare il gruppo specificato

```
user = super().save(commit)
```

#ottengo un riferimento all'utente

```
g = Group.objects.get(name="Lettori")
```



#cerco il gruppo che mi interessa

```
g.user_set.add(user)
```


#aggiungo l'utente al gruppo

```
return user
```


#restituisco quello che il metodo padre di questo metodo avrebbe restituito.

Banalmente, per i bibliotecari

```
class CreaUtenteBibliotecario(UserCreationForm):  
  
    def save(self, commit=True):  
        user = super().save(commit)  
        g = Group.objects.get(name="Bibliotecari")  
        g.user_set.add(user)  
        return user
```



Dato che da specifiche solo un admin (quindi membro dello staff) può iscrivere un Bibliotecario, possiamo creare una Create User View in biblio3/views.py protetta con l'access mixin chiamato "permission required". La variabile "permission_required" ci permette di specificare la condizione necessaria. L'utente admin è l'unico ad avere (di default) privilegi di staff.



```
from django.contrib.auth.mixins import PermissionRequiredMixin  
class BiblioCreateView(PermissionRequiredMixin, UserCreateView):  
    permission_required = "is_staff"  
    form_class = CreaUtenteBibliotecario
```

gestione/templates/gestione/home.html

```
{% block content %}
|
<h2>Cosa vuoi fare oggi?</h2>

<br>

<div class="list-group">
  <a href="{% url 'gestione:cercalibro' %}" class="list-group-item list-group-item-info">Cerca un libro</a>
  <a href="{% url 'gestione:listalibri' %}" class="list-group-item list-group-item-info">Lista dei titoli</a>
  <a href="{% url 'gestione:situation' %}" class="list-group-item list-group-item-info">La mia situazione</a>
</div>

<br>

{% if user.is_staff %}
<br>
<a href="{% url 'registerb' %}" class="list-group-item list-group-item-danger">Iscrivi un bibliotecario</a>
<br>
{% endif %}

{% if "Bibliotecari" in user.groups.all.0.name or user.is_staff %}
<br>
<a href="{% url 'gestione:situationb' %}" class="list-group-item list-group-item-warning">Situazione Biblioteca</a>
<a href="{% url 'gestione:crealibro' %}" class="list-group-item list-group-item-warning">Aggiungi un libro</a>
<a href="{% url 'gestione:creacopia' %}" class="list-group-item list-group-item-warning">Aggiungi una copia</a>
<br>
{% endif %}

{% endblock %}
```

Menu
personalizzati in
funzione della
tipologia di utente

Cosa vuoi fare oggi?

Cerca un libro

Lista dei titoli

La mia situazione

Iscrivi un bibliotecario

Situazione Biblioteca

Aggiungi un libro

Aggiungi una copia

Protezione Views con criterio di appartenenza a Gruppi

Ora che abbiamo diverse categorie (o meglio **gruppi**) di utenti, possiamo usare dei decorator appositi per controllare che l'utente che richiede una certa funzionalità abbia l'autorizzazione necessaria. Genericamente:

```
def has_group(user):  
    return user.groups.filter(name='nome_gruppo').exists()  
  
from django.contrib.auth.decorators import user_passes_test  
  
@user_passes_test(has_group)  
def my_view(request):  
    ...
```

Ed i mixins per i Gruppi?

Django qui offre molto meno rispetto a quello che abbiamo visto per l'autenticazione.

Esistono però moduli esterni ed aggiuntivi che ci permettono di avere queste funzionalità. In particolare, si consiglia l'uso di **django_braces**:



```
pipenv install django-braces
```

<https://django-braces.readthedocs.io/en/latest/>

django-braces

Questa libreria esterna fornisce molte classi da utilizzare come access mixins normalmente non presenti nella distribuzione standard di django. Non è strettamente necessaria, ma il suo utilizzo ci permette di risparmiare un sacco di codice per risolvere problemi ricorrenti (come per esempio, i test di appartenenza a uno o più gruppi).



Alcuni mixin di braces: **GroupRequiredMixin**, SuperUserRequiredMixin, MultiplePermissionRequiredMixin....

```
from braces.views import GroupRequiredMixin
class BiblioSituationView(GroupRequiredMixin, ListView):
    group_required = ["Bibliotecari"]
    model = Libro
    template_name = "gestione/situationb.html"

class BiblioDetailView(GroupRequiredMixin, DetailView):
    group_required = ["Bibliotecari"]
    model = Libro
    template_name = "gestione/detailb.html"

class CreateLibroView(GroupRequiredMixin, CreateView):
    group_required = ["Bibliotecari"]
    title = "Aggiungi un libro alla biblioteca"
    form_class = CreateLibroForm
    template_name = "gestione/create_entry.html"
    success_url = reverse_lazy("gestione:home")

class CreateCopiaView(CreateLibroView):
    title = "Aggiungi una Copia ad un libro"
    form_class = CreateCopiaForm
```

Osservazioni

- Per passare un test di appartenenza ad un gruppo, occorre essere loggati.
- L'admin, in quanto "is_superuser" passa sempre tutti i test di appartenenza ai gruppi, pur non essendo mai stato lui esplicitamente aggiunto a tali gruppi.



- Quello che succede se un utente "Lettore" prova a fare il bibliotecario o l'admin dipende dal modo in cui abbiamo protetto le rispettive view.
 - Esempio: loggato come lettore
 - CASO A: Provo ad andare in localhost:8000/registrar/ **#solo per admin: registra #bibliotecario**
 - CASO B: Provo ad andare in localhost:8000/gestione/situationb/ **#per bibliotecari: #situazione prestiti di tutti i libri**

← → ↻ ⓘ localhost:8000/registerb/

403 Forbidden

Pagina di Login

Si prega di compilare i campi necessari

[Lista dei Libri](#) [Cerca un Libro](#)

Si è arrivati qui in quanto non si dispone dei permessi necessari per il servizio scelto!

Username*

Password*

Esegui login