

## introduzione

web framework free e open source python-based che usa il pattern architettonico **model-template-view**

**framework:** applicazione "vuota", "bozza" facilmente personalizzabile per una qualsiasi esigenza reale

**web framework:** offre strumenti modulari per tutte le funzionalità basiliari nella realizzazione di una web app (autenticazione, gestione utenti, connessione a db, ...)

**pattern architettonico:** struttura che sottende al framework e impone un certo rigore nel workflow del lavoro

### model-template-view

**model:** definizione di tabelle su db

**template:** pagina HTML

**view:** business logic e gestione delle risposte del server

## setup

### pipenv

**ambiente virtuale:** strumento che mantiene separate le dipendenze richieste da diversi progetti creando ambienti isolati per ciascuno

```
mkdir project_dir  
cd project_dir
```

```
pipenv install django #provvede anche a creare il venv
```

```
pipenv shell #attiva il venv
```

### startproject

**django-admin** comando per le attività amministrative

```
django-admin startproject project_name ./
```

#struttura del progetto creato:

```
project_dir  
└── manage.py  
└── Pipfile  
└── Pipfile.lock  
└── project_name  
    ├── __init__.py  
    ├── asgi.py  
    ├── settings.py  
    ├── urls.py  
    └── wsgi.py
```

**settings.py** controlla le preferenze

**urls.py** gestisce quello che django dovrà restituire a seguito di una richiesta

**wsgi.py** aiuta a servire eventuali webpages

**manage.py** per eseguire diversi comandi di django es. creare un server locale o una nuova app

**local server:** python manage.py runserver

django usa il concetto di **app** e **project** per mantenere il codice pulito, leggibile e modulare

differenze tra app e project:

**app:** applicazione web con funzionalità particolari e specifiche, può essere usata in molteplici project

**project:** collezione di configurazioni e app per una applicazione web specifica

### settings.py

**BASE\_DIR** → path assoluta da usare in **TEMPLATES**, **STATIC\_ROOT**, ...

**TEMPLATES** → definisce, tra le altre, la location dei template

**STATIC\_ROOT** → path assoluta alla cartella nella quale **collectstatic** aggiunge static file

**static files** → file statici aggiuntivi come immagini, js, css eccetera

○ **collectstatic** → comando per aggregare tutti gli static file provenienti dalle app del progetto in un'unica dir (usato per deployment ottimizzati)

**STATIC\_URL** → url da usare per accedere agli static file in **STATIC\_ROOT**

**STATICFILES\_DIRS** → path relative agli static files provenienti dalle app

**DEBUG** → se true mostra su schermo gli errori nel dettaglio, se false si limita a dichiarare lo "status code"

**ALLOWED\_HOST** → scopo: convalidare header host HTTP di una richiesta; strettamente legato a **DEBUG**

○ con **DEBUG=False**, **ALLOWED\_HOST=[ ]** django non si avvia

**INSTALLED\_APPS** → si inseriscono tutte le app presenti nel progetto (nostre e da librerie)

### templates

sono l'approccio più comune per generare HTML in modo dinamico

un template contiene le parti statiche dell'output HTML desiderato + alcune sintassi speciali per i contenuti dinamici

in django, sono file HTML con contenuto statico e dinamico

```
#settings.py  
#alla chiave DIRS nel dizionario in TEMPLATES assegnamo  
"DIRS" : [os.path.join(BASE_DIR, 'templates')],  
# inoltre crea cartella templates all'interno del progetto
```

per specificare la rappresentazione dei dati si usano:

**template tags** - controllano il rendering dei template

**template variables** - rimpiazzate da valori a tempo di rendering

**template filters** - modificano le variabili a tempo di rendering

```
{% tag %}  
{{ variable }}  
{{ variable|filter }}  
  
#esempio di template tag  
{% block <block_name> %} #definisce un blocco di contenuto
```

i template si possono combinare:

```
{% extends "<template_name>" %}
```

spesso accade che alcuni elementi compaiano su più pagine - con django è sufficiente "estendere" un template su un file html, che basterà completare inserendo il contenuto delle tag del template per una pagina completa

○ convenzione template naming: usa **base.html** per il template base

○ esempio di uso di template e t.tags →

```
<!— base.html —>  
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    <meta charset="UTF-8">  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <meta http-equiv="X-UA-Compatible" content="ie=edge">  
    <title>{% block title %}{% endblock %}</title>  
  </head>
```

```

<body>
  <!-- other static HTML content -->
  {% block content %}

  {% endblock %}
</body>
</html>

<!-- estendiamo su home.html -->
{% extends 'base.html' %}

{% block title %}Home Page{% endblock %}

{% block content %}
<h1>La mia home page</h1>
<p>
  Lorem ipsum dolor sit amet, consectetur adipisicing elit,
  sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
</p>
{% endblock %}

```

#### static tags e static files

```

{% load static %} viene usato per caricare static tag
{% static 'path' %} sintassi per caricare static file nella dir static

#settings.py
STATIC_URL = '/static/'
STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')]

```

#### views.py

file costituito da **viste** - ciascuna è responsabile della business logic e della risposta del server in seguito ad una particolare richiesta  
la vista ritorna un oggetto `HttpResponse` oppure solleva un'eccezione come `HTTP404`  
il file non è creato automaticamente → lo facciamo in `project_name` (quella interna, non la dir contenitore) e inseriamo:

```

from django.http import HttpResponse

def home(request):
    return HttpResponse("Hello world! Questa è la home page.")

```

ci sono molte funzioni e classi per ritornare `HttpResponse` e mostrare template a schermo, tra cui:

`render()` (fz) → combina un template con dei parametri passati tramite l'oggetto `context`

✓ `context` → dizionario che mappa keyvalue usato nei template per generare contenuto dinamico

`TemplateView (C)` → mostra su schermo uno specifico template con il `context` contenente i parametri presenti nell'url

per mostrare dati:

`ListView (C)` → `self.object_list`\* conterrà la lista degli oggetti (solitamente ma non sempre un `queryset`) sulla quale la view sta operando

`DetailView (C)` → `self.object`\* conterrà l'oggetto sul quale la view sta operando

✓ \* attributo accessibile dal template

#### urls.py

contiene una lista nella quale ciascun elemento rappresenta un url collegato ad una vista specifica (funzione o classe che sia) e ad un nome utilizzabile nei template

#inseriamo questo codice in urls.py:

```
from django.contrib import admin
```

```

from django.urls import path
from . import views #dalla stessa dir di urls.py importa views.py

urlpatterns = [
    path('', views.home, name='home'),
    path('admin/', admin.site.urls),
]

```

primo url pattern:

' ' → si accede alla view senza passare argomenti all'url  
`views.home` → esegue la funzione `home` di `views.py`  
`name='home'` → assegna il nome `home` alla path (per url reversing)

#### primi passi

proviamo ad aggiungere un url verso una pagina `home`:

```

# views.py

# ...
def home_page(request):
    response = "Benvenuto nella homepage di prova!"
    return HttpResponseRedirect(response)

# urls.py
from django.urls import path, re_path
# ...
urlpatterns = [
    # ...
    path("home/", views.home_page, name="homepage")
]

```

creiamo degli alias per far puntare più indirizzi alla stessa home page → usiamo le **regex**  
per gli url django: usiamo `re_path`:

```

urlpatterns = [
    # ...
    re_path(r"^\$|^\$/|^home/\$", views.home_page, name="homepage")
]

```

#### django logging system

attivo di default, complesso, per modificarlo occorre impostare dei parametri in `settings.py`

<https://docs.djangoproject.com/en/4.0/topics/logging/>

in generale, la sintassi è `logger.<severità del messaggio>(msg)`

uno dei parametri impostabili è la severità oltre la quale vengono visualizzati i messaggi su `stdout`

per provare:

```

# ...
import logging
logger = logging.getLogger(__name__)

def home_page(request):
    response = f"Benvenuto nella homepage, {str(request.user)}"

    if not request.user.is_authenticated:
        logger.warning(f'{str(request.user)} non è autenticato!')

    return HttpResponseRedirect(response)

```

```
formato di default del logger: timestamp request_type url_path protocol status_response bytes_sent  
[11/Feb/2022 09:52:53] "GET /admin/login/?next=/admin/ HTTP/1.1" 200 2215
```

#### response codes (in breve)

1XX	Messaggio informativo
2XX	Success!
3XX	Redirection
4XX	Fail (colpa client)
5XX	Fail (colpa server)

<https://developer.mozilla.org/en-US/docs/Web/HTTP>Status>

404 → basta inserire da browser un url "non matchabile" con le regole elencate in urlpatterns

500 → errore lato server es. syntax error nel codice

in release è possibile fare **intercettare** da django queste risposte per fornire all'utente pagine personalizzate di errore  
in debug ci conviene leggere attentamente gli errori!

#### richieste GET

ammettono parametri specificati tramite URL

es. url\_path?param1=val1&param2=val2...

nella view function posso ottenerli tramite request.GET["param\_name"]

esempio: implementiamo una funzione che ritorni i parametri alla vista

# views.py

```
def elenca_params(request):  
    response = ""  
    for k in request.GET:  
        response += request.GET[k]  
  
    return HttpResponse(response)
```

```
# urls.py  
# aggiungo agli urlpatterns  
path('elenca_parametri/', views.elenca_params, name="params")
```

altro modo per passare parametri **evitando le stringhe <url\_path>p=v&... (solitamente risultato di azioni client/server side e non scritte manualmente)**:

tramite url path:

→ diventa possibile raggiungere tutte le combinazioni valide imposte dalle regole espresse in urlpatterns

```
path('url_path/<type:param_name>/ ... ', view_func, name='alias')  
#esempi  
path('url_path/<int:eta>', v_foo, name='a') #un solo param intero "eta"  
path('url_path/<str:nome>/<int:eta>', v_foo, name='a')  
#esempi di path validi:  
# localhost:8000/url_path/Mario/23  
# localhost:8000/url_path/Carlo/18
```

posso accedere a questi param. come **argomenti in ingresso alla funzione**:

```
# views.py  
def welcome_path(request,nome,eta):  
    return HttpResponse(f"Si chiama {nome} ed ha {str(eta)} anni")
```

```
# urls.py  
urlpatterns=[  
    # ...  
    path("welcome/<str:nome>/<int:eta>/", views.welcome_path, name="welcomepath"),  
    re_path(r"^\w{a-zA-Z0-9}+$", views.welcome_user, name="welcomeuser")  
]  
  
# ● http://127.0.0.1:8000/welcome_path/Mario Rossi/23/  
# ○ OK  
# ● http://127.0.0.1:8000/welcome_path/5/23/  
# ○ OK  
# ● http://127.0.0.1:8000/welcome_path/Mario/Ciao/  
# ○ KO: "Ciao" non è un intero!  
# ● http://127.0.0.1:8000/welcome_path/Mario/2.3/  
# ○ KO: "2.3" non è un intero!
```

vi è un vero e proprio **type enforcement!** mismatch nel tipo di dato prevede mismatch anche nelle regole di generazione

#### DTL

finora abbiamo visto come restituire stringhe tramite l'oggetto HttpResponseRedirect

vogliamo pagine formattate seguendo HTML, evitando però di passarle come le stringhe precedenti...

**scomodo, difficile da estendere e riutilizzare, viola la policy DRY, decisamente non generico**

→ si usano i **template!** scritti unendo HTML e DTL, **Django Template Language**, basato su **blocchi** anziché tag

file DTL:

appaiono come sorgenti statici, ma sono **risolti dinamicamente lato server e spediti come "risultato statico"** all'utente  
comoda interfaccia sia alla parte di Python, sia per accedere ai parametri delle richieste HTTP  
estendibili evitando ripetizioni

riprendiamo i template, estendiamo nuovamente base.html su templates/baseext.html

```
{% extends 'base.html' %}  
  
{% block title %} {{title}} {% endblock %}  
  
{% block content %}
```

```
<h1>  
    {% if user.is_authenticated %}  
        Ciao, {{ user.username }}!  
    {% else %}  
        Ciao, guest!  
    {% endif %}  
</h1>
```

```
{% for i in lista %}  
  
    <p> {{ i }} </p>
```

```
{% endfor %}  
  
{% endblock %}
```

# views.py

```
def hello_template(request):  
    # importa  
    # from datetime import datetime  
    # from django.shortcuts import render  
    ctx = {  
        "title" : "Hello Template",
```

```

"lista" : [ datetime.now(), datetime.today().strftime('%A'), datetime.today().strftime('%B')]
}

return render(request, template_name="baseext.html", context=ctx)

# urls.py
path("hellotemplate/", views.hello_template, name="hellotemplate")

template composition H3
{% extends "template.html" %}
ma anche {% include "template.html" %}

"copia-incolla" del codice HTML già processato/renderizzato all'interno di un altro template (l'argomento è detto subtemplate)
<!-- <root_pr>/templates/includefooter.html -->

<footer>
Link utili <br>
<p><a href="http://localhost:8000/welcome/Mario/25/">Welcome name, age!</a></p>
</footer>

<!-- baseext.html -->

{% extends 'base.html' %}

{% block title %} {{title}} {% endblock %}

{% block content %}

<h1> #...
</h1>

{% for i in lista %} #...
{% endfor %}

{% include "includefooter.html" %}

{% endblock %}

```

alternativa DTL per blocco link dinamico (per rappresentare un url):

```

<footer>
Link utili <br>
<p><a href="{% url 'welcomepath' nome='Mario' eta=25 %}">Welcome name, age!</a></p>
</footer>

```

**! ☑** tramite include, i sub-template vengono pre-renderizzati - NON C'È UNO STATO CONDIVISO TRA TEMPLATE E SUB-T!

se ridefinisco blocchi (nel template che usa extends) definiti nel padre, questi vengono sovrascritti completamente

```

<!-- base.html -->
<!-- ... -->
{% block content %}
    contenuto 1
{% endblock %}

<!-- baseext.html -->
{% extends "base.html" %}
<!-- ... -->
{% block content %}
    contenuto 2
{% endblock %}

```

```

        to 2
        {% endblock %}

baseext.html renderizza contenuto 2!

```

per ereditare il contenuto del blocco padre: uso all'interno del blocco ridefinito {{block.super}}

### lettura parametri url tramite DTL

richieste GET da url\_path?nome=value1&eta=value2...:

```

<p>
    {{ request.GET.nome }}
</p>
<p>
    {{ request.GET.eta }}
</p>

<!-- per controllarli tutti ... -->
{% for key, value in request.GET.items %}
    {{ key }} = {{ value }}
{% endfor %}

```

alternativa:

```

<p> {{ request.resolver_match.kwargs.nome }} </p>
<p> {{ request.resolver_match.kwargs.eta }} </p>

```

### risorse statiche

tutto ciò che non viene generato dinamicamente in django/modificabile dall'utente  
gestione non sempre banale, e differenziata a seconda dell'impostazione DEBUG

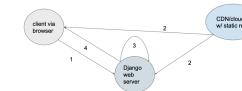
**! ☑** i template risiedono nella stessa macchina in cui è presente la logica python, per le risorse statiche django non fa assunzione

comunemente sono caricate *on-demand* e risiedono in servizi cloud

CDNs (content delivery networks)

altri host remoti

possibile soluzione:



1. **contatto del client** al django web server, con pagina contenente sia el. dinamici che statici (es. immagine)

2. l'immagine va **servita** da un server apposito

oppure (eventualmente) il web server ha una **cache** per la risorsa

3. viene **combinata** con gli altri elementi dinamici

4. la pagina viene **rispedita** al client

[servire risorse statiche in production](#)

### servire risorse static con DEBUG=True

```

# settings.py

INSTALLED_APPS = [
    ...
    'django.contrib.staticfiles',
]

```

```

]

STATIC_URL = 'static/'
STATICFILES_DIRS = [os.path.join(BASE_DIR, "static")]

cartella static a livello root (come templates)
nel blocco head di un template DTL+HTML:

{% load static %}

utilizzo:


<link rel="stylesheet" type="text/css" href="{% static '/css/style.css' %}">

```

considerazioni:



## generic vs app template/static files H2

sia i template che i file statici possono essere  
ad uso e consumo di una app specifica  
generici → uso e consumo di più app, o del solo progetto di root

⚠ attenzione ai conflitti a livello di **naming**

## template engine H2

i template **non sono** semplici html, bensì oggetti parte di una complessa struttura interna che si occupa (sempificando) di generare dinamicamente il contenuto HTML da mostrare su browser → **il template engine**  
un progetto django può essere configurato con 0+ template engine  
offerti backend integrati per **DTL** (django template language) e per la popolare alternativa **ninja2** (altri integrati tramite librerie esterne)  
django definisce un API standard (indipendente da backend) per  
**caricamento**: *loading* - trovare il template che fa riferimento ad un certo identificatore e preprocessarlo (solitamente compil. in memoria)  
**rendering**: interporare il template con il **context** e ritornare la stringa risultante

### configurazione H3

```
# settings.py

TEMPLATES = [
{
    'BACKEND': 'django.template.backends.django.DjangoTemplates',
    'DIRS': [],
    'APP_DIRS': True,
    'OPTIONS': {
        # opzioni
    }
}
```

BACKEND: path ad una classe del template engine che implementa le API - i template integrati sono

```

django.template.backends.django.DjangoTemplates
django.template.backends.jinja2.Jinja2

DIRS: lista di dirs (in ordine di ricerca) dove l'engine deve cercare i source dei template
APP_DIRS: definisce se l'engine deve cercare template dentro alle app installate (ogni backend definisce un nome convenzionale per le subdir nelle app dove sono memorizzati i template)
OPTIONS: setting specifici per il backend
funzioni per caricare template definite dal modulo django.template.loader:
get_template(template_name, using=None) → carica un template e ritorna un oggetto Template (il tipo esatto dipende dal backend, ognuno ha la propria classe Template)
la fz prova ogni engine dato (in ordine) finché uno non ha successo
per restringere a un solo engine, si usa using
select_template(template_name_list, using=None) → analogo, ma accetta una lista di nomi di template
```

| template non trovato solleva **TemplateDoesNotExist**  
| template trovato ma con sintassi invalida solleva **TemplateSyntaxError**  
**funzione** che automatizza il processo di render:  
**render\_to\_string(template\_name, context=None, request=None, using=None)** → carica un template e chiama immediatamente il suo metodo **render()**  
**template\_name**: singolo o lista, viene usato get o select rispettivamente  
**context**: dizionario usato come template context  
**request**: una **HttpRequest** opzionale disponibile durante il processo di rendering

| la shortcut **render()** chiama **render\_to\_string()** e inserisce il risultato in una **HttpResponse** adatta per essere ritornata da una view

### template language H3

i template possono contenere  
**variabili** H4  
{{ variable }} (combinazione di alfanum e underscore, **non inizia con underscore**)  
sostituite con valori all'elaborazione del template  
{{ variable.attribute }} per accedere agli attributi di una variabile  
quando l'engine incontra un punto, cerca (in ordine)  
1. dizionario  
2. attributo o metodo  
3. indice numerico

se il valore è **callable** viene chiamato senza argument, ed il risultato della chiamata diventa il valore sul template

per variabili inesistente viene inserito il valore di **string\_if\_invalid** (vuota di default)

si possono modificare tramite **template filters**

{{ variable|filter[|another\_filter]... }}

es. lower, truncatewords:30 (con argument)

argument con spazi vanno quotati

elenco completo dei builtin filter

### tag H4

{% tag %}

output testuale, operatori condizionali, loop, caricamento informazioni esterne

alcuni richiedono tag di apertura e chiusura

tag più comuni:

```
% for %
{ if %, { elif %, { else %
% extends %
% block %
#{ comment #} / {# comment %} {# endcomment %}
```

## modificare oggetti del db tramite view H2

finora lo abbiamo fatto solo come admin, evidentemente vogliamo poterlo fare anche come utenti normali

il modulo `django.views.generic.edit` offre i seguenti strumenti:

`FormView`: view che mostra un form

in caso di errori nella compilazione, lo mostra di nuovo con gli errori

se compilato con successo, reindirizza ad un nuovo url

`CreateView`: mostra un form per **creare** un oggetto

rимотра in caso di errori, salva altrimenti

usato per inserire nuove entry in una specifica tabella del db, necessita di

`model`: il model al quale aggiungere l'oggetto

`template_name`: il template usato per il render

`fields`: i campi da compilare per inserire la entry nel database (in lista o tupla)

`success_url`: url al quale si viene reindirizzati a salvataggio avvenuto

`UpdateView`: mostra un form per **modificare** un oggetto esistente

rимотра in caso di errori, aggiorna altrimenti

`DeleteView`: mostra una pagina di conferma e cancella un oggetto esistente

le ultime due usano un codice per la view e una path identici a quello di `CreateView`, salvo per

`Delete` non deve specificare `fields`

entrambe le classi usano un url con **identificativo** per accedere all'oggetto specifico (vedi esempio)

⚠️ l'app `soci` viene creata successivamente - seguire la procedura giusta per integrarla nel progetto di prova da `startapp`

`#soci/views.py`

```
from django.urls import reverse_lazy
from django.views.generic.edit import CreateView
```

```
class PersonaCreate(CreateView):
    model = Persona
```

```
    fields = ('nome', 'cognome', 'ruolo')
```

```
    template_name = 'soci/persona_create.html'
```

```
    success_url = reverse_lazy('soci:persona-list')
```

`#soci/urls.py`  
#path da aggiungere

```
path('persona-create', views.PersonaCreate.as_view(), name='persona-create')
```

```
## soci/templates/soci/persona_create.html body: #
<form method="post">{# csrf_token %
{{ form.as_p }}
<input type="submit", value="Save">
</form>
```

`csrf_token`: Cross Site Request Forgery → strumento usato come protezione da attacchi csrf, obbligatorio nei form con metodo POST

per renderizzare il form, si passa l'oggetto `form` con la sintassi delle template variables → presenti i seguenti metodi:

```
as_p: ogni <input> viene inserito in un <p>
as_table: ogni <input> viene inserito in una cella di tabella all'interno di un <tr> (bisogna aggiungere <table> nel file)
as_ul: ogni <input> viene inserito in un <li> all'interno di una <ul> (bisogna aggiungere <ul> nel file)
# path per ipotetiche pagine di update e delete, con l'uso di id
path('persona-update/<int:pk>', views.PersonaUpdate.as_view(), name='persona-update')
path('persona-delete/<int:pk>', views.PersonaDelete.as_view(), name='persona-delete')
```

## startapp H2

per creare un'app e integrarla nel progetto è necessario seguire una procedura precisa:

`python manage.py startapp <nome_app>` nella cartella del progetto

django genera in automatico la cartella prepopolata

```
soci
├── __init__.py
├── admin.py
├── apps.py
└── migrations
    └── __init__.py
        ├── models.py
        └── tests.py
└── views.py
```

`admin.py`: qui si registrano i `model` (tabelle del db) da includere nella sezione `admin` di django

`apps.py`: principali configurazioni dell'app

`models.py`: `models` dell'app

`tests.py`: test per l'app

`views.py`: specifiche dell'app

`migrations`: contiene le migrazioni di db dell'app (vedi sezione `models` H2)

⚠️ non vengono generati la dir `templates` e `urls.py` non essendo indispensabili

buona pratica per evitare conflitti di naming è strutturare le cartelle dei templates delle app in questo modo:

```
app_name
└── templates/
    └── app_name/
        └── template.html
```

per tenere traccia dell'app nel progetto django bisogna "registrarla" in `settings.py` > `INSTALLED_APPS`

aggiungi (dall'esempio) `'soci.apps.SociConfig'`

adopera i comandi `makemigrations/migrate` per far sì che vengano create le tabelle sul db (vedi sezione `models` H2)

crea file `urls.py`

buona pratica assegnare alla variabile `app_name` il nome dell'app → crea un namespace specifico ed evita conflitti con altri url usati nei template (potenzialmente omonimi)

à questo punto la sintassi per assegnare un url django ad un elemento HTML diventa `{% url '<app_name>:path' %}`

`#soci/urls.py`

```
from django.urls import path
```

```

from . import views
app_name = 'soci'

#urls.py

from django.urls import include, path
from . import views

urlpatterns = [
    # ...
    path('soci/', include('soci.urls')),
    # ...
]

```

## models

strumento usato da django per **strutturare i dati sul db**

contiene informazioni sui campi e i comportamenti dei dati memorizzati  
solitamente ogni model viene mappato su **una singola tabella** del db

ogni model è classe che eredita da `django.db.models.Model`

ogni attributo rappresenta un campo del db - le classi assegnate agli attributi fanno parte del modulo `models` e **definiscono il tipo di dato che verrà salvato**

django genera automaticamente API di accesso al db

```
#soci/models.py
from django.db import models

class Persona(models.Model):
    nome = models.CharField(max_length=50)
    cognome = models.CharField(max_length=50)
```

il modello `Persona` crea la tabella seguente sul db

```
CREATE TABLE soci_persona (
    "id" serial NOT NULL PRIMARY KEY,
    "nome" varchar(50) NOT NULL,
    "cognome" varchar(50) NOT NULL
);
```

il nome della tabella (`soci_persona`) viene generato automaticamente come il campo `id` (la **primary key** del model), ma entrambi sono sovrascrivibili

## migrazioni

modo in cui django estende le modifiche fatte sui model al database

è una sorta di vcs dello schema db

modi più comuni per gestirle

`makemigrations`: crea nuove migrazioni basate sui cambiamenti rilevati, ovvero genera un file nel quale vengono elencate creazioni e modifiche di model

`migrate`: il comando che **applica** le migrazioni (in base al file generato in precedenza)

django offre tipi che rappresentano **relazioni tra tabelle**:

`ForeignKey` (n a 1): richiede due arg posizionali

la classe al quale il model si relaziona

`on_delete`: accetta `models.CASCADE`, `models.PROTECT` (più usati)

```

related_name: nome usato dall'oggetto messo in relazione a quello con al foreign key per accedervi
da un'istanza di Ruolo sarà possibile accedere ad una lista contenente tutte le Persone aventi un determinato ruolo con
la sintassi istanza_ruolo.persone.all()

#soci/models.py

#in Persona:
ruolo = models.ForeignKey(Ruolo, on_delete=models.PROTECT, related_name='persone')

#aggiungo il model Ruolo:
class Ruolo(models.Model):
    titolo = models.CharField(max_length=30)

class Meta:
    verbose_name_plural = 'Ruoli' #per bypassare il default di aggiungere s alla fine della parola
```

`ManyToManyField`: richiede la classe con la quale il model si relaziona

`django reifica` in automatico (crea tabella intermedia)

`OneToOneField`: concettualmente simile a fk con `unique=True`, ma l'accesso attraverso `related_name` ritorna un **oggetto singolo** usato principalmente come pk di un model che "estende" un altro model

## createsuperuser

interfaccia admin automatica che legge i metadati dei model per fornire un'interfaccia rapida ed efficiente di gestione dei contenuti

```
python manage.py createsuperuser
accesso da 127.0.0.1:8000/admin
```

di default la pagina di amministrazione contiene solo i model Groups e User (autenticazione di default di django) → per **aggiungere i propri model** bisogna modificare `admin.py`:

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

```
admin.site.register(Persona)
```

## class based views

metodi alternativi per implementare view come **oggetti anziché come funzioni**; le più usate:

`DetailView`: visualizza i dati relativi ad un'entry di una specifica tabella del db

`default` ritorna un context con l'oggetto `object` (istanza dell'entry)

usata per visualizzare **una sola entry** - bisogna specificare (es. tramite `url param`) a quale entry ci stiamo riferendo

```
#soci/views.py
from django.views.generic.detail import DetailView

class PersonaDetail(DetailView):
    model = Persona
    template_name = 'soci/persona_detail.html'
```

```
#soci/urls.py
path('persona/<int:pk>', views.PersonaDetail.as_view(), name='soci_detail'),
```

```
{# soci/templates/soci/persona_detail.html in body: #}
```

```

{{ object.pk }} - {{ object.nome }} {{ object.cognome }}

ListView: ritorna tutte le entry della tabella
def ritorna un context con l'oggetto object_list

#soci/views.py

from django.views.generic.detail import ListView

class Personalist(ListView):
    model = Persona
    template_name = 'soci/persona_list.html'

#soci/urls.py

path('persona-list/', views.Personalist.as_view(), name='persona-list'),

## soci/templates/soci/persona_list.html in body: #

{% for persona in object_list %}
    {{ persona.nome }} {{ persona.cognome }}<br>
{% endfor %}

```

## Models - approfondimenti

La sezione contiene il codice dell'esercitazione **biblioteca**

### filosofia ORM

Object Relational Mapping - usata da django per la gestione di dati strutturati, tramite librerie interne che "traducono" oggetti (istanze di classi) in entità fruibili (es. **dati in un db**)

le classi stesse rappresentano le tabelle, le istanze le entry

**migrazione:** il processo di traduzione python ↔ db query

approccio molto comodo:

non serve scrivere query né legarsi alle scelte implementative del db

le classi ORM sono ben integrate con il resto del codice

possiamo sfruttare ereditarietà per massimizzare il riuso, e personalizzare a piacere aggiungendo metodi e META-informationi

migrazioni come strumento aggiuntivo per evitare danni

### esercitazione - setup

```

# crea dir progetto
mkdir biblioproject && cd biblioproject
# crea venv ed entra
pipenv install django
pipenv shell
# crea progetto biblio
django-admin startproject biblio .
# crea app gestione
python manage.py startapp gestione

# crea cartelle templates
mkdir templates
mkdir -p gestione/templates/gestione

# aggiungi base.html in templates es. copiandolo
cp path_to_template/base.html templates/base.html

```

```

# biblio/settings.py

# aggiungi:
import os
# template dirs:
DIRS : [os.path.join(BASE_DIR,'templates')], 
# aggiungi app gestione in INSTALLED_APPS:
'gestione',

# copia urls
cp biblio/urls.py gestione/urls.py

# biblio/urls.py

# aggiunta di include all'import
from django.urls import path, include
# aggiunta di pattern
path('gestione/',include('gestione.urls')) 

# gestione/urls.py
from gestione import views
app_name = 'gestione'

# aggiunta di pattern
path('',views.home, name='home')

# gestione/views.py
from django.http import HttpResponseRedirect
#...
def home(request):
    return HttpResponseRedirect('Hello world!')

proseguiamo creando un semplice db:
Libro con Titolo, Autore, Pagine, data ultimo prestito
1 copia per libro
per il momento senza utenti che prendono in prestito/restituiscono nulla

# gestione/models.py
from django.db import models

class Libro(models.Model):
    titolo = models.CharField(max_length=200)
    autore = models.CharField(max_length=50)
    pagine = models.IntegerField(default=100)
    data_prestito = models.DateField(default=None)

    def __str__(self):
        out = self.titolo + " di " + self.autore
        if self.data_prestito == None:
            out += " attualmente non in prestito"
        else:
            out += " in prestito dal " + str(self.data_prestito)

    # preparo la migrazione app-level
    python3 manage.py makemigrations gestione
    # Migrations for 'gestione':
    #     - migrations\0001_initial.py
    #         - Create model Libro

```

```
# rendo effettiva la migrazione
python3 manage.py migrate
```

### inserire elementi nel database

programmaticamente → tramite python, nel setup iniziale, a tempo di import/export dati, routine di manutenzione,...

python shell con python3 manage.py shell

```
>>> from gestione.models import Libro
>>> q = Libro.objects.all()
>>> type(q)
<class 'django.db.models.query.QuerySet'>
>>> l = Libro()
>>> l.titolo = "Promessi Sposi"
>>> l.autore = "Alessandro Manzoni"
>>> print(l)
Promessi Sposi di Alessandro Manzoni attualmente non in prestito
>>> l.save()
100
>>> from django.utils import timezone
>>> l.data_prestito = timezone.now()
>>> print(l)
Promessi Sposi di Alessandro Manzoni in prestito dal 2022-02-15 00:31:06.183132+00:00
>>> l.save()
101
>>> l = Libro.objects.all()
>>> QuerySet [Libro: Promessi Sposi di Alessandro Manzoni in prestito dal 2022-02-15]
>>> lg = Libro.objects.get(pk=1)
>>> lg
<Libro: Promessi Sposi di Alessandro Manzoni in prestito dal 2022-02-15>
```

possiamo anche scrivere funzioni apposite:

```
# root_prj/initcmds.py
```

```
from gestione.models import Libro
from django.utils import timezone
from datetime import datetime

def erase_db():
    print("Cancello il DB")
    Libro.objects.all().delete()

def init_db():
    if len(Libro.objects.all()) != 0:
        return

    def func_time(off_year=None, off_month=None, off_day=None):
        tz = timezone.now()
        out = datetime(tz.year-off_year, tz.month-off_month,
                      tz.day-off_day, tz.hour, tz.minute, tz.second)
        return out

    #se è vuoto lo inizializzo
    #può essere letto da fonti esterne, files, altri DB etc...
    libridict = {
        "autori" : ["Alessandro Manzoni", "George Orwell", "Omero", "George Orwell", "Omero"],
        "titoli" : ["Promessi Sposi", "1984", "Odissea", "La Fattoria degli Animali", "Iliade"],
        "pagine" : [832,328,414,141, 263],
        "date" : [ func_time (y,m,d) for y in range(2) for m in range(2) for d in range(2) ]
    }

    for i in range (5):
        l = Libro()
        for k in libridict:
            if k == "autori":
                l.autore = libridict[k][i]
            elif k == "titoli":
                l.titolo = libridict[k][i]
            elif k == "pagine":
                l.pagine = libridict[k][i]
            else:
                l.data_prestito = libridict[k][i]
        l.save()

    print("DUMP DB")
    print (Libro.objects.all()) #controlliamo
```

per farle eseguire ad es. allo startup del server, invece di uno script a parte possiamo inserirle in root\_prj/urls.py:

```
# aggiungo import
from .initcmds import * # definizione di erase_db e init_db

urlpatterns = [
    ...
]

erase_db()
init_db()
```

tramite admin console → GUI web

deve esistere almeno un admin → **createsuperuser**

aggiungi in gestione/admin.py:

```
from .models import Libro
admin.site.register(Libro)
```

tramite views e function views → modifiche scatenate da richieste client-side

### Function Views ed integrazione con DTL

le operazioni viste finora sono di amministrazione - come tali vanno svolte da admin e **non devono essere esposte al pubblico**  
vediamo come unire le richieste GET su HTTP (attivate quando un utente cerca una risorsa) con le interazioni db tramite python  
→ **response function views**

creiamo una view con template per mostrare il contenuto del db

```
# gestione/views.py

from django.http import HttpResponseRedirect
from django.shortcuts import render
from .models import Libro

def lista_libri(request):
    templ = "gestione/listalibri.html"

    ctx = {
        "title" : "Lista di Libri",
        "listalibri" : Libro.objects.all()
    }

    return render(request, template_name=templ, context=ctx)
```

```
# gestione/urls.py

from django.urls import path
from .views import *

app_name = "gestione"

urlpatterns = [
    path("listalibri/", lista_libri, name="listalibri")
]
```

!-- gestione/templates/gestione/listalibri.html -->

{% extends "base.html" %}

{% block head %} {% endblock %}

```

{%
    block title %}{{title}}{% endblock %}

{%
    block content %

```

<center>

<h1>{{title}}</h1>

{% if listalibri.count > 0 %}

<p> Ci sono ben {{listalibri.count}} libri in questa biblioteca! </p>

<ul>

{% for l in listalibri %}

<li> {{ l }} </li>

{% endfor %}

{% else %}

<p>Non ci sono libri!</p>

{% endif %}

</center>

{% endblock %}

ora proviamo ad isolare i libri "mattone" (300+ pg) → tramite una funcview che riutilizza lo stesso templ e visualizza solo i mattoni  
usiamo un'operazione di filtro sul QUerySet del nostro Model, per poi passare la lista risultante tramite il dizionario di contesto

```

# gestione/views.py

MATTONE_THRESHOLD = 300

def mattoni(request):
    templ = "gestione/listalibri.html"

    lista_filtrata = Libro.objects.filter(pagine__gte=MATTONE_THRESHOLD)
    # lista_filtrata = Libro.objects.exclude(pagine__lt=MATTONE_THRESHOLD)

    ctx = {
        "title" : "Lista di mattoni",
        "listalibri" : lista_filtrata
    }

    return render(request,templ,ctx)

```

i QuerySet.filter() possono essere arbitrariamente complessi, in generale consentono di verif. non solo uguaglianze :

<nome\_attributo>\_\_[gt,lt,gte,lte, ...] = value, ...

altre operazioni prevedono ad esempio l'ordinamento:

```

qs.filter(attr__[gt,lt,gte,lte, ...]=val, ...)
qs.order_by('attr')
qs.order_by('-attr') #senso decrescente
qs.order_by('attr')[2] # i primi due risultati
qs.filter(attr_iexact=val) #case insensitive
qs01 = qs0 | qs1 #unione di queryset

```

filter dovrebbe bastare per praticamente tutto, non fosse così si possono comunque usare raw queries se siamo sicuri  
che il db sottostante **di fatto lo sia** e ne conosciamo l'implementazione

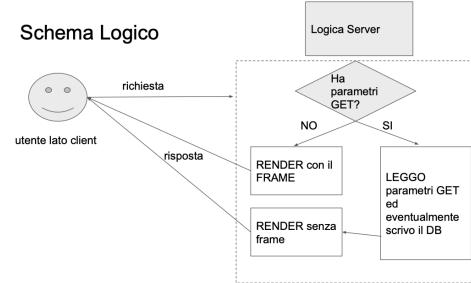
raw queries in django models:

```

for l in Libro.objects.raw("SELECT * FROM gestione_libro WHERE pagine >=%s", [MATTONE_THRESHOLD]):
    print(l)

```

|  
| **le raw query restituiscono oggetti RawQuerySet, sostanzialmente ≠ e privi ad esempio del metodo count usato nel templ**  
| **aggiunta di entries nel db** HA  
| form html che prenda tramite textField gli attributi del libro  
| pulsante che, se premuto, riporta alla stessa pagina ma con i parametri GET impostati tramite url  
| la function deve distinguere tra  
| primo arrivo → mostra il form  
| form già riempito (GET params) → crea il libro



```

def crea_libro(request):
    message = ""

    if "autore" in request.GET and "titolo" in request.GET:
        aut = request.GET["autore"]
        tit = request.GET["titolo"]
        pag = 100

        try:
            pag = int(request.GET["pagine"])
        except:
            message = "Pagine non valide. Inserite pagine di default."

        l = Libro()
        l.autore = aut
        l.titolo = tit
        l.pagine = pag
        l.data_prestito = timezone.now()

        try:
            l.save()
            message = "Creazione libro riuscita! " + message
        except Exception as e:
            message = "Errore nella creazione del libro " + str(e)

    return render(request,template_name="gestione/creolibro.html",context={"title":"Crea autore", "message" : message})

```

```

{%
    extends "base.html" %

{%
    block head %} {%
    endblock %

{%
    block title %}{{title}}{%
    endblock %

{%
    block content %

<center>
<h1>{{title}}</h1>

```

```

        out += "copia non scaduta"
    return out

class Meta:
    verbose_name_plural = "Copie"

a questo punto migriamo:

python3 manage.py makemigrations gestione
# Migrations for 'gestione':
#   - gestionne/migrations/0002_alter_libro_options_remove_libro_data_prestito_copia.py
#       ~ Change Meta options on libro
#       - Remove field data_prestito from libro
#       + Create model Copia

python3 manage.py migrate
# Operations to perform:
#   Apply all migrations: admin, auth, contenttypes, gestione, sessions
#   Running migrations:
#     Applying gestione.0002_alter_libro_options_remove_libro_data_prestito_copia... OK

| questa è andata a buon fine, ma altre sono potenzialmente distruttive → sono salvate in script (simil-vcs)
| esempi di migrazioni potenzialmente distruttive:
aggiunta di campi senza valori di default o non-nullable: admin dovrà risolvere manualmente i conflitti
cambio di primary key: potenziale perdita dei dati salvati
→ occorrerebbe esportare il db prima di questo tipo di migr, e poi scrivere script ad hoc per tradurre i dati vecchi nella nuova struttura
nel nostro caso: non abbiamo perso dati, ma fallirebbe il comando di popolamento → va modificato initcmds.py

|
```

## conclusioni

siamo in grado di accedere r/w a un db anche tramite azioni utente, in particolare abbiamo implementato tutti i metodi CRUD

DRY completamente violato: fin troppo copypaste → vedremo le Class Views and Forms

access control: vogliamo lasciare accesso r/w a tutti? → introdurremo Access control Mixin and Permission

## Modelli e Migrazioni - qualche dettaglio in più

| La sezione contiene il codice dell'esercitazione **biblioteca2**

### esercitazione - setup

stessi passaggi di **esercitazione - setup**

creo progetto biblio2, creo app gestione, creo modello Libro con gli stessi metodi ed attributi, creo i metodi di gestione per riempire il db  
rendiamo più realistico il sistema bibliotecario, introducendo il concetto di **Copia**

modifichiamo 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 __str__(self):
        out = f'{self.titolo} di {self.autore} ha {str(self.copie.all().count())} copie'
        return out

```

```

class Meta:
    verbose_name_plural = "Libri"

```

```

class Copia(models.Model):
    data_prestito = models.DateField(default=None,null=True)
    scaduto = models.BooleanField(default=False)
    libro = models.ForeignKey(Libro, on_delete=models.CASCADE, related_name="copie")

```

```

    def __str__(self):
        out = f'Copia di {self.libro.titolo} di {self.libro.autore}: '
        if self.scaduto:
            out += "copia scaduta"
        else:

```

```
        date = [ func_time(y,m,d) for y in range(2) for m in range(2) for d in range(2) ]
```

```

for i in range(5):
    l = Libro()
    for k in libridict:
        if k == "autori":
            l.autore = libridict[k][i]
        if k == "titoli":
            l.titolo = libridict[k][i]
        if k == "pagine":
            l.pagine = libridict[k][i]
    l.save()
    for d in date:
        c = Copia()
        c.libro = l
        c.data_prestito = d
        c.save()

```

ora abbiamo un db **quasi coerente**: vogliamo un controllo periodico sulla scadenza dei libri → possiamo usare un thread periodico lanciato come operazione di inizializzazione

```

# initcmds.py
def controllo_scadenza():

    MAX_PRESTITO_GIORNI = 15

    print("Controllo copie scadute in corso ... ")
    for l in Libro.objects.all():
        s0 = l.copie.filter(scaduto=False).exclude(data_prestito=None)
        for c in s0:

```

```

dt = datetime(timezone.now().year,timezone.now().month,timezone.now().day).date()
if (dt - c.data_prestito) > timedelta(days = MAX_PRESTITO_GIORNI):
    c.scaduto = True
    c.save()
    print(c)

def start_controllo_scadenza(check_time_in_seconds):
    Timer(check_time_in_seconds, controllo_scadenza).start()

```

## Class Based Views - approfondimento

finora abbiamo visto come sfruttare l'intero design pattern di django:

**models:** ORM e migrazioni

**views:** funzioni che regolano la logica delle webapp

**templates:** lato presentazione

dal punto di vista funzionale non ci sono problemi, mentre è fortemente **limitata la comodità di sviluppo**

rispondono alle richieste sempre con function views non sempre consente di sfruttare la policy DRY, ma python è **object-oriented** - le classi solitamente sono un buon approccio per riutilizzare codice

→ rispondiamo alle richieste tramite classi anziché tramite funzioni

	FBV	CBV
cosa usa	funzioni	metodi/classi/oggetti
riutilizzabilità	bassa (molta ripetizione)	alta (poca o nessuna ripetizione tra view impare)
decorators	supportati	sì e no (si passa per i mixin)
DB access di base	molto codice esplicito e difficilmente riutilizzabile	riduzione sostanziale - possibile inheritance da moduli
generiche op.	scelta migliore quando le logiche da implementare aumentano di complessità	partendo dalle view di django, alcune op. diventano implementabili
custom		

| FBV e CBV non sono mutualmente esclusive, bensì meccanismi complementari!

| La sezione contiene il codice dell'esercitazione **CBVprj**

```

# project setup
mkdir CBVprj && cd CBVprj

pipenv install django
pipenv shell

django-admin startproject cbvprj
cd cbvprj # poiché senza ./ alla fine del cmd:
    # precedente la struttura è la seguente:

```

```

# CBVprj
#   ├── cbvprj
#   |   ├── __init__.py
#   |   ├── asgi.py
#   |   ├── settings.py
#   |   ├── urls.py
#   |   └── wsgi.py
#   └── manage.py
# └── Pipfile

```

```

python3 manage.py runserver
^C
python3 manage.py migrate

```

```

python3 manage.py startapp iscrizioni

mkdir -p templates iscrizioni/templates/iscrizioni
cp path_to_other_templates/base.html templates/base.html

python3 manage.py createsuperuser

```

```

# ##### #
# cbvprj/settings.py
# #####
# INSTALLED_APPS
'iscrizioni',
# TEMPLATES["DIRS"]
os.path.join(BASE_DIR,'templates')

# #####
# iscrizione/admin.py
# AGGIUNGIAMO I MODELLI (DOPO AVERLI CREATI)
# #####

```

```

from .models import *
admin.site.register(Studente)
admin.site.register(Insegnamento)

# #####
# iscrizioni/models.py
# CREIAMO I MODELLI
# #####
class Studente(models.Model):
    name = models.CharField(max_length=50)
    surname = models.CharField(max_length=50)

    def __str__(self):
        return "ID: " + str(self.pk) + ": " + self.name + " " + self.surname

class Meta:
    verbose_name_plural = "Studenti"

```

```

class Insegnamento(models.Model):
    titolo = models.CharField(max_length=50)
    studenti = models.ManyToManyField(Studente,default=None)

    def __str__(self):
        return "ID: " + str(self.pk) + ": " + self.titolo

```

```

class Meta:
    verbose_name_plural = "Insegnamenti"

# #####
# cbvprj/initcmds.py
# funzioni chiamate in urls.py per inizializzazione db
# #####
from iscrizioni.models import Studente, Insegnamento

def erase_db():
    print("Cancello il DB")
    Studente.objects.all().delete()
    Insegnamento.objects.all().delete()

def init_db():
    # LOGSEQ

```

```
python3 manage.py makemigrations iscrizione
python3 manage.py migrate
```

## Metodi CRUD sui DB tramite CBV - ListView

operazioni di base: creazione, lettura, aggiornamento, rimozione di entry

django mette a disposizione delle classi "View" per i metodi CRUD, appartenenti ai moduli `django.views.generic.*`  
→ si crea una classe che **estende** una di queste, specificando a quale **Model** ci si riferisce e quale **template** usare

...e basta! abbiamo finito!

```
# iscrizioni/views.py

from django.views.generic.list import ListView
from .models import *
```

# Create your views here.

```
class ListaStudentiView(ListView):
    model = Studente
    template_name = "iscrizioni/lista_studenti.html"
```

```
# iscrizioni/views.py

from .views import *

urlpatterns = [
    path("listastudenti/", listaStudentiView.as_view(), name="listastudenti")
]
```

`as_view()`: metodo della classe `View` che restituisce un **oggetto funzione** contenente i meccanismi della view (la funzione `path` si aspetta di lavorare con funzioni)

```
!—— iscrizioni/lista_studenti.html —>
{% extends 'base.html' %}

{% block title %} Dump del DB Studenti {% endblock %}
{% block content %}

{% for p in object_list %}
<ul>
    {{ p }}
</ul>
{% endfor %}

{% endblock %}
```

`object_list`: variabile di contesto ritornata dalla `ListView` che contiene il `QuerySet` operato sul modello specificato

```
# function view alternativa in formato funzione (decisamente più verbosa)

def lista_studenti_function(request):
    lista = Studente.objects.all()
    templ = "iscrizioni/lista_studenti.html"
    ctx = {"object_list": lista}

    return render(request, templ, ctx)
```

**filtraggio e variabili di contesto**

di default non possiamo pre-processare liberamente il queryset risultante dal model...anche se esistono dei filtri appositi  
get\_queryset: per operare sulla tabella indicata dall'attributo `model`  
get\_context\_data: per aggiungere **variabili di contesto**

```
# iscrizioni/views.py

# elenca insegnamenti con almeno un iscritto
class ListaInsegnamentiAttivi(ListView):
    model = Insegnamento
    template_name = "iscrizioni/insegnamenti_attivi.html"

    def get_queryset(self):
        return self.model.objects.exclude(studenti__isnull=True)

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['titolo'] = "Insegnamenti Attivi"
        return context
```

`titolo` è una variabile di contesto aggiunta, mentre `object_list` è sempre **ereditata**  
altre variabili ereditate:  
[`'paginator'`, `'page_obj'`, `'is_paginated'`, `'object_list'`, `'insegnamento_list'`, `'view'`]  
`view`: permette di **accedere a metodi e attributi** del nostro derivato di `ListView` → possiamo definire ulteriori logiche e **chiamarle lato template**  
esempio: cbv che elenca gli studenti e restituisce il numero totale di iscrizioni agli insegnamenti

```
# iscrizioni/views.py

class ListaStudentiIscritti(ListView):
    model = Studente
    template_name = "iscrizioni/studenti_iscritti.html"

    def get_model_name(self):
        return self.model._meta.verbose_name_plural

    def get_context_data(self, **kwargs):
        ctx = super().get_context_data(**kwargs)
        ctx["titolo"] = "Lista studenti con iscrizioni"
        return ctx

    def get_totale_iscrizioni(self):
        count = 0
        for i in Insegnamento.objects.all():
            count += i.studenti.all().count()
        return count
```

```

<!-- iscrizioni/templates/iscrizioni/studenti_iscritti.html -->

{% extends 'base.html' %}

{% block title %} {{titolo}} {% endblock %}

{% block content %}

<h1> Lista di {{ view.get_model_name }} </h1>

e

<p> Il numero totale di iscrizioni è {{view.get_totale_iscrizioni}}

{% endblock %}

```

grazie a view ho moltissime possibilità di espansione, ma non infinite: ad es. non posso chiamare view methods con parametri dal template

#16146 closed New feature (wontfix)

Calling functions with arguments inside a template

  
 Resolution: -- won't fix  
 Status: now -> closed  
 Django actually prevents this by design. The idea is to keep most of the logic in views and keep the template language as simple as possible (so that it can easily be used by non-tech-savvy designers, for example). Logic from the view must be moved to the template if you want to pass arguments to methods that are accessed from within templates. Data should be calculated in views, then passed to template for display. (<https://docs.djangoproject.com/en/1.3/topics/templates/#accessing-methods>)  
 So you have to do that in the view. Writing a custom template tag or filter might also be of good help. This is a common problem, so don't hesitate to ask on the django-users mailing list for more advice.

## CreateView

django.views.generic.edit.CreateView

in aggiunta a model e template:

fields: per indicare quali attributi si voglia permettere di impostare al client  
 success\_url: per l'url di redirect in caso di scrittura a buon fine

# iscrizioni/views.py

```

class CreateStudenteView(CreateView):
    model = Studente
    template_name = 'iscrizioni/crea_studente.html'
    fields = '__all__'
    success_url = reverse_lazy("iscrizioni:listastudenti")

```

# iscrizioni/urls.py

```
path('creastudente/', CreateStudenteView.as_view(), name='creastudente')
```

## reverse, reverse\_lazy

from django.urls import reverse, reverse\_lazy

eseguono un reverse lookup, ossia invece di specificare un url hardcoded, possiamo usare l'alias dell'url (tipicamente app\_name:url\_name)  
 il motore di django si assicura di creare un pattern in grado di matchare con il redirect voluto  
 necessario in quanto più string pattern possono matchare con lo stesso url, inoltre permette anche più manutenibilità (non devo cambiare il valore di success\_url se cambio quello in urls.py)

reverse si usa in metodi e funzioni: restituisce una stringa

reverse\_lazy restituisce un oggetto complesso: si usa quando assegno valori a variabili/attributi → viene eseguito tardivamente, e confonderà i porta a errori tipo Reverse not found

legato al funzionamento di import e l'init degli attributi di una classe: python inizializza prima di importare e risolvere gli url patterns → se non ritardassi il reverse lookup, il nome alias rischia di non trovare corrispondenza

```

# alternativa equivalente:

class CreateStudenteView(CreateView):
    model = Studente
    template_name = "iscrizioni/crea_studente.html"
    fields = "__all__"
    success_url = reverse_lazy("iscrizioni:listastudenti")

    #metodo ereditato dalla classe padre
    def get_success_url(self):
        return reverse("iscrizioni:listastudenti")

```

<!-- iscrizioni/templates/iscrizioni/crea\_studente.html -->

```

{% extends 'base.html' %}

{% block title %} Crea Studente {% endblock %}

{% block content %}

<h1> Crea Studente </h1>

<form method="post">{{ csrf_token }}
  {{ form.as_p }}
  <input type="submit" value="Save">
</form>

{% endblock %}

```

i campi della tabella sono inclusi dalla context variable form, da inserire in un <form> HTML  
 specifichiamo che la submission avviene tramite richiesta POST (non GET)  
 form esplicita i fields da noi definiti e li renderizza in vari modi (vedi sotto)

per renderizzare il form, si passa l'oggetto form con la sintassi delle template variables → presenti i seguenti metodi:

as\_p: ogni <input> viene inserito in un <p>

as\_table: ogni <input> viene inserito in una cella di tabella all'interno di un <tr> (bisogna aggiungere <table> nel file)

as\_ul: ogni <input> viene inserito in un <li> all'interno di una <ul> (bisogna aggiungere <ul> nel file)

	GET	POST
BACK button/Reload	Harmless	Data will be re-submitted (the browser user that the data are about to be re-submitted)
Bookmarked	Can be bookmarked	Cannot be bookmarked
Cached	Can be cached	Not cached
Encoding type	application/x-www-form-urlencoded	application/x-www-form-urlencoded or data. Use multipart encoding for binary data. Use multipart encoding for binary data.
History	Parameters remain in browser history	Parameters are not saved in browser history
Restrictions on data length	Yes, when sending data, the GET method adds the data to the URL; and the length of a URL is limited (maximum URL length is 2048 characters)	No restrictions
Restrictions on data type	Only ASCII characters allowed	No restrictions. Binary data is also allowed

	GET	POST
Security	GET is less secure compared to POST because data sent is part of the URL Never use GET when sending passwords or other sensitive information!	POST is a little safer than GET because data are not stored in browser history or in the URL
Visibility	Data is visible to everyone in the URL	Data is not displayed in the URL

```

o a request.GET["param_name"] corrisponde request.POST["param_name"]

{% csrf_token %} H4
meccanismo di sicurezza che siamo obbligati a specificare - reso possibile perché in settings.py è listato
django.middleware.csrf.CsrfViewMiddleware nella variabile MIDDLEWARE

```

evita il Cross Site Reference Forgery:

una volta che la potenziale vittima fa il login al sito, viene salvato nel browser il cookie di autenticazione. a questo punto un hacker può indurre a visitare url per modificare il DB, cambiando però i parametri inseriti a suo piacimento

il csrf previene ciò associando la prima parte della richiesta ad un token autogenerato, inviando la stringa al client e tenendosi il token lato server → se un client compromesso prova a modificare il DB mandando un token non presente/non mandando token, il server blocca le operazioni

vediamo che se cambiamo dalla browser console il token o ad es. il limite massimo di caratteri da inserire, otteniamo una pagina di errore:



## DetailView H3

query ad una tabella partendo dalla pk → otteniamo un context con la variabile object attraverso cui possiamo scegliere i campi da visualizzare (e come) nel template

```
# iscrizione/views.py

class DetailInsegnamentoView(DetailView):
    model = Insegnamento
    template_name = 'iscrizioni/insegnamento.html'

# iscrizioni/urls.py

path('insegnamento/<pk>', DetailInsegnamentoView.as_view(), name='insegnamento')
```

la chiave viene specificata direttamente nell'url (vedi sopra)

```
<!-- iscrizioni/templates/iscrizioni/insegnamento.html -->

{% extends 'base.html' %}

{% block title %} {{object.titolo}} {% endblock %}

{% block content %}
```

```
<h1>Dettagli di {{object.titolo}}</h1>
```

```
<br>

{% if object.studenti.all.count == 0 %}
```

Nessuno studente iscritto

```
{% else %}

    {% for s in object.studenti.all %}

        <li> {{s.name}} {{s.surname}} </li>

    {% endfor %}

    {% endif %}

    {% endblock %}
```

## UpdateView H3

accessibilità simile a DetailView (pk nell'url)  
modifica simile a CreateView (stesso metodo per restituire form lato template, richiesta POST, token csrf)

attributi richiesti: template\_name, model, success\_url, fields

```
class UpdateInsegnamentoView(UpdateView):
    model = Insegnamento
    template_name = "iscrizioni/edit_insegnamento.html"
    fields = "__all__"

    def get_success_url(self):
        pk = self.get_context_data()["object"].pk
        return reverse("iscrizioni:insegnamento", kwargs={'pk': pk})
```

## DeleteView H3

come Create ed Update per ottenere la pk dell'elemento da cancellare  
occorre comunque il form di conferma

```
# iscrizioni/views.py

# iniziamo a sfruttare l'ereditarietà
class DeleteEntitaView(DeleteView):
    template_name = 'iscrizioni/cancella_entry.html'

    def get_context_data(self, **kwargs):
        ctx = super().get_context_data(**kwargs)
        entita = "Studente"
        if self.model == Insegnamento:
            entita = "Insegnamento"
        ctx['entita'] = entita
        return ctx

    def get_success_url(self):
        if self.model == Studente:
            return reverse('iscrizioni:listastudenti')
        else:
            return reverse('iscrizioni:listainsegnamenti')

class DeleteStudenteView(DeleteEntitaView):
    model = Studente
```

```

class DeleteInsegnamentoView(DeleteEntitaView):
    model = Insegnamento

Esercizi bonus H3
combinare views ereditate da django.views.generic.* con input passati tramite url path

# iscrizioni/urls.py

path('studente/<str:surname>', ListStudenteBySurname.as_view(), name="studente")

# iscrizioni/views.py

class ListStudenteBySurname(ListaStudentiView):
    # eredita gli attributi model e template
    def get_queryset(self):
        arg = self.kwargs['surname'] # leggiamo l'arg. da url
        qs = self.model.objects.filter(surname__iexact=arg) #case insensitive
        return qs

(logseq)

Django Forms H2
abbiamo visto come sfruttare l'intero design pattern di django (models, views, templates)

abbiamo risparmiato molto codice usando Class Based Views - approfondimento H2
CBV offrono "in regalo" la variabile di contesto form, che permette di evitare il tediosissimo boilerplate HTML per i form

⚠ i problemi sorgono quando le operazioni si complicano oltre le CRUD su una singola tabella
due tabelle, List View su un form con criteri particolari,... (form e validazione personalizzata) → non si scappa
es. la ricerca di studenti per nome e/o cognome (vedi Esercizi bonus H3 n.2)
abbiamo dovuto codificare campo per campo in HTML/DTL ed esplicitare la logica
soluzione: django forms
from django import forms
estendendo la classe posso definire
1. campi editabili personalizzati
2. condizioni aggiuntive di validazione per tali campi

⚠ La sezione contiene il codice dell'esercitazione mysite_frameCBV
inizializziamo il progetto e partiamo con l'app polls, modelli e tabelle per Question, Choice (anche superuser, static dirs, migrazioni)

# polls/models.py

class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

    def was_published_recently(self):
        return self.pub_date >= timezone.now() - datetime.timedelta(days=1)

    def __str__(self):
        return self.question_text

class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE, related_name="choices")
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)
    is_correct = models.BooleanField(default=False)

def __str__(self):
    return self.choice_text

def _init_db(self):
    if Question.objects.count() > 0:
        return
    data = None

# taken from: https://opentdb.com/api.php?amount=50&category=18&type=multiple
# print(os.getcwd())
filepath = os.path.join(Path(os.getcwd()), "static", "questions", "questions.txt")
# print(filepath)
# print(os.path.exists(filepath))

with open(filepath) as f:
    data = f.read()

if data is None:
    print("Error in populating database tables!")
    return

data = json.loads(data)["results"]

for i, d in enumerate(data):
    question = d["question"]
    choices = []
    choices.append(d["correct_answer"])
    choices.extend(d["incorrect_answers"])
    random.shuffle(choices)
    date = timezone.now() - timedelta(days=i % 10)
    q = Question(question_text=html.unescape(question), pub_date=date)
    q.save()
    # print("DOMANDA " + str(i) + " " + str(q))
    for k, j in enumerate(choices):
        c = Choice(choice_text=html.unescape(j), votes=k, question=q)
        if j == d["correct_answer": c.is_correct = True
        c.save()
        # print("Risposta " + str(c))

# static/questions/questions.txt
{"response_code":0,"results":[{"category":"Science: Computers","type":"multiple","difficulty":"easy","question":"Which company founded in 1975, is the world's largest manufacturer of personal computers?"}]}

iniziamo con due list view base:
polls/: home con le 20 domande più recenti (ognuna punta con un link alla sua pagina "detail")
polls/pk/detail: detailview sulla domanda identificata dalla pk, mostrando le relative scelte (e voti)

# polls/views.py

class IndexViewList(ListView):
    model = Question
    template_name = 'polls/index.html'

    def get_queryset(self):
        return self.model.objects.order_by('-pub_date')[:20]

class DetailQuestion(DetailView):

```

```

model = Question
template_name = "polls/detail.html"

# polls/urls.py
app_name = 'polls'

urlpatterns = [
    path('', IndexViewList.as_view(), name='index'),
    path('<pk>/detail/', DetailQuestion.as_view(), name='detail')
]

i template rimangono estremamente simili ai precedenti
view successiva: un form di ricerca delle domande → sarebbe utile poter cercare per argomento
ci serve
1. una search string per la parola chiave
2. sapere dove cercare (in Question o Choice)

tipicamente si crea un file apposito che raccoglie i Form personalizzati dell'app - lo chiamiamo forms.py
un form personalizzato estende django.forms.Form

campi editabili di un form
in django.forms.(fields|widgets).py trovo tutti i possibili campi con cui popolare il form (ed i widget o compon. grafici associati a ciascuno)
django docs - form fields
per ora, scegliamo CharField per la keyword e ChoiceField per scegliere dove cercare (lasciamo i widget di default)

# polls/forms.py

from django import forms

class SearchForm(forms.Form):
    CHOICE_LIST = [("Questions", "Search in Questions"),
                  ("Choices", "Search in Choices")]

    search_string = forms.CharField(label="Search String", max_length=100, min_length=3, required=True)
    search_where = forms.ChoiceField(label="Search Where?", required=True, choices=CHOICE_LIST)

# polls/urls.py
path('search/', search, name='search'),
path('searchresults/<str:sstring>/<str:where>/', SearchResultsList.as_view(), name='searchresults')

# polls/views.py

# ...
from .forms import *
# ...

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")
            # importa redirect da django.shortcuts
            return redirect("polls:searchresults", sstring, where)
    else:
        form = SearchForm()

    return render(request, template_name='polls/searchpage.html', context={'form':form})

osservazioni:
se la richiesta è GET, istanziamo senza parametri un SearchForm e lo passiamo come context variable al template
nonostante form ora è stata creata da noi, segue le stesse regole di formattazione viste con i form di default, dunque:
1. as_p, as_table, as_ul funzionano
2. usiamo sia il metodo POST che il token csrf
3. lo apriamo e chiudiamo noi con tag HTML e pulsante submit
<!-- polls/templates/polls/searchpage.html -->
{% extends "base.html" %}

{% block title %} Search Page Form {% endblock %}

{% block content %}

<h1>Search Page</h1>

<form action="{% url 'polls:search' %}" method="POST"> {% csrf_token %}
    {{form.as_p}}
    <input type="submit" value="Search"> </form>

{% endblock %}

per ogni elemento creato nel form personalizzato
è associata una label (creata come elemento HTML aggiuntivo)
appare in forms.py come variabile (nome che riempie il campo name del rispettivo elemento HTML, ottenibile tramite request.POST)
gli elementi HTML hanno ulteriori attributi, i quali sono la traduzione degli ulteriori parametri specificati
tornando a search(request): se il metodo è POST, in request.POST abbiamo tutti i dati che servono (tutti quelli con attr name specificato)
non accediamo direttamente al dizionario, ma lo passiamo al costruttore del nuovo SearchForm
# ...
form = SearchForm(request.POST)
# ...

in questo modo il sistema legge e valida i dati passati al form → form.cleaned_data.get(...) restituisce un input "sanitizzato"
con redirect possiamo dirottare i campi letti verso l'url resolver (in questo caso chiamiamo la view risolta tramite
'polls:searchresults' che ammette 2 parametri)
creiamo una list view, con override di get_queryset per ottenere i risultati filtrati:

class SearchResultsList(ListView):
    model = Question
    template_name = "polls/searchresults.html"

    def get_queryset(self):
        sstring = self.request.resolver_match.kwargs["sstring"] where = self.request.resolver_match.kwargs["where"]
        if "Question" in where:
            qq = Question.objects.filter(question_text__icontains=sstring)
        else:
            qc = Choice.objects.filter(choice_text__icontains=sstring)
            qq = Question.objects.none()
            for c in qc:
                qq |= Question.objects.filter(pk=c.question_id)
        return qq

il template è sempre simile, poiché ereditando da ListView ∃ la context variable object_list che contiene oggetti di tipo Question (che si visualizzare)

```

## integrazione tra Models e Forms HO

come abbiamo visto per le CBV, possiamo associare un Form ad una particolare tabella (se definita come un oggetto Python tra i model) in questo modo possiamo

1. creare campi di scelta in funzione di querysets
2. creare ModelForm, ovvero Form legati a dei modelli in cui non dobbiamo necessariamente specificare i campi editabili dall'utente e a cui 2. possiamo dare direttive arbitrarie sul rendering (widget) e sulle regole di validazione

### ModelChoiceField HA

creiamo una view che permetta di votare ad una question (selezionata con pk) (raggiungibile tramite link dalle due ListView)

il voto avviene tramite form, contenente un campo a scelta multipla per selezionare la risposta

al submit avviene il redirect ad una detailview apposita che informa se la risposta è corretta

incrementa il campo votes della risposta

```
# polls/forms.py
from django.shortcuts import get_object_or_404

# ...

class VoteForm(forms.Form):
    answer = forms.ModelChoiceField(queryset=None, required=True, label="Select your answer!")

    def __init__(self, pk, *args, **kwargs):
        super().__init__(*args, **kwargs)
        q = get_object_or_404(Question, pk=pk)
        self.fields['answer'].queryset = q.choices.all()
```

answer è legato ad un modello → le possibili scelte non sono hardcoded, bensì dipendono dinamicamente da un queryset (e quindi da un modello)

```
# polls/views.py
# ...

def vote(request, pk):
    if request.method == "POST":
        form = VoteForm(data=request.POST, pk=pk)
        if form.is_valid():
            answer = form.cleaned_data.get("answer")
            return redirect("polls:votecasted", pk, answer.choice_text)
        else:
            q = get_object_or_404(Question, pk=pk)
            form = VoteForm(pk=pk)
            return render(request, template_name="polls/vote.html", context={"form": form, "question": q})
```

rispetto all'esempio precedente, abbiamo istanziato il form **dovendo esplicitare il parametro costruttivo aggiuntivo**, il quale serve per recuperare le possibili scelte relative alla domanda

il redirect porta poi ad una detailview che usa pk per recuperare la domanda giusta e fa override di get\_context\_data per aggiungere le informazioni richieste

```
# polls/views.py
# ...

class VoteCastedDetail(DetailView):
    model = Question
    template_name = "polls/votecasted.html"

    def get_context_data(self, **kwargs):
        ctx = super().get_context_data(**kwargs)
```

```
        answer = self.request.resolver_match.kwargs["answer"]
        ctx["answer"] = answer
        correct = ctx["object"].choices.all().get(is_correct=True)
        if answer in correct.choice_text:
            ctx["message"] = "Right Answer!"
        else:
            ctx["message"] = "Wrong Answer! " + "The right answer was " + str(correct.choice_text)

        try:
            c = ctx["object"].choices.all().get(choice_text=answer)
            c.votes += 1
            c.save()
        except Exception as e:
            print("Impossible to update vote value " + str(e))

        return ctx
```

### ModelForm e CBV HA

finora abbiamo provato a legare un campo del form ad un queryset → quindi ad un modello  
legare l'intero form ad un model ci permette invece di avere controllo aggiuntivo su **come l'utente specifica i dati in ingresso**  
esistono i ModelForm: seguono le regole già viste, ma danno la possibilità di creare meta info per specificare a quale tabella sono collegati una volta specificato il model e l'attributo fields (come visto per le CreateView), **non occorre specificare i campi del form**

1. esempio 1: CBV per la creazione di Question CreateQuestionForm
  - una domanda deve avere minimo 5 caratteri
2. esempio 2: creazione di Choice CreateChoiceForm
  - non posso inserirla in domande con già 4 risposte
  - esiste 1 sola risposta corretta su 4

```
# polls/forms.py
# ...

class CreateQuestionForm(forms.ModelForm):
    description = "Create a new Question!"

    def clean(self):
        if len(self.cleaned_data["question_text"]) < 5:
            self.add_error("question_text", "Error: question text must be at least 5 characters long")

        return self.cleaned_data

    class Meta:
        model = Question
        fields = "__all__"
        widgets = {
            'pub_date': forms.DateInput(format='(%d/%m/%Y'),
                                   attrs={'class': 'form-control', 'placeholder': 'Select a date', 'type': 'date'})}

class CreateChoiceForm(forms.ModelForm):
    description = "Create choices for a question"

    def clean(self):
        q = get_object_or_404(Question, pk=self.cleaned_data["question"].id)

        choices = q.choices.all()
        choices_false = choices.filter(is_correct=False)

        if choices.count() == 4:
            self.add_error("question", "Error: question already has four options")
        elif choices.count() == 3:
```

```

if choices_false.count() == 3 and not self.cleaned_data["is_correct"]:
    self.add_error("is_correct", "Error: exactly one choice must be correct")

if choices.filter(is_correct=True).count() == 1 and self.cleaned_data["is_correct"]:
    self.add_error("is_correct", "Error: This question already has a correct answer")

return self.cleaned_data

class Meta:
    model = Choice
    fields = "__all__"

entrambi
ereditano da forms.ModelForm
hanno una var. opzionale description
hanno specificato in Meta gli attributi model e fields (simile alle CreateView)
operano un override di clean per implementare validazione input aggiuntiva

| quella std è stata eseguita prima, con i dati pre-validati disp in self.cleaned_data (dizionario con chiavi gli attributi del model)
| gli errori aggiunti in clean (se scatenati) compariranno nel render del browser e impediranno la sottomissione dei dati POST
rispetto all'altro modeform, questo ha un parametro in più nelle META informazioni: widgets → dizionario che associa ad un attributo del model (stringa) un widget diverso dal default

# polls/views.py

class CreateQuestionView(CreateView):
    template_name = "polls/createentry.html"
    form_class = CreateQuestionForm
    success_url = reverse_lazy("polls:index")

class CreateChoiceView(CreateView):
    template_name = "polls/createentry.html"
    form_class = CreateChoiceForm

    def get_success_url(self):
        ctx = self.get_context_data()
        pk = ctx["object"].question_pk
        return reverse("polls:detail", kwargs={"pk": pk})

# polls/urls.py

path('createquestion/', CreateQuestionView.as_view(), name='createquestion'),
path('createchoice/', CreateChoiceView.as_view(), name='createchoice'),


<!-- polls/templates/polls/createentry.html --&gt;
{% extends "base.html" %}

{% block title %} {{ view.form_class.description }} {% endblock %}

{% block content %}

&lt;h1&gt; {{ view.form_class.description }} &lt;/h1&gt;

&lt;form method="POST"&gt; {% csrf_token %}
    {{ form.as_p}}
    &lt;input type="submit" value="Create"&gt;
</pre>

```

```

</form>
{% endblock %}

```

### Crispy forms

per quanto django offre strumenti veloci ed efficienti per realizzare form con pochissimo sforzo, non permette di curarne l'aspetto  
→ django-crispy-forms è un'app che permette di interagire con proprietà dei form direttamente dal backend, controllando così il comportamento di rendering dei django form in modo elegante e DRY  
per installarlo: (dall'interno del pipenv) pip install --upgrade django-crispy-forms

```

# settings.py

INSTALLED_APPS = (
    # ...
    'crispy_forms',
)

```

I'app offre supporto integrato per diversi framework CSS, chiamati **template packs**

bootstrap (default)  
bootstrap3  
bootstrap4  
uni-form → standardizza markup e stile dei form e lo modella con CSS  
foundation → accessibile esternamente attraverso crispy-forms-foundation

per specificare il template pack:

```

# settings.py

CRISPY_TEMPLATE_PACK = '<nome-template-pack>'

```

### crispy filter

- aggiungi tag {% load crispy\_form\_tags %}
- aggiungi il filtro |crispy alla context variable form

```

{% load crispy_form_tags %}

<form action='home/' method="POST">
    {{ form|crispy }}
</form>

```

per quanto utile, è molto simile nell'uso a as\_p,... e dunque non permette una customizzazione consistente

### {% crispy %}

I'app implementa la classe FormHelper che va a definire il comportamento di rendering del form, controllando attributi e layout del form → la scrittura di HTML diviene minima, e le specifiche avvengono tutte in forms e views

per gli esempi useremo il seguente ModelForm relativo a Persona dal tutorial\_project

python

```

class PersonaCrispyForm(forms.ModelForm):

    helper = FormHelper()
    helper.form_id = 'persona-crispy-form'
    helper.form_method = 'POST'
    helper.add_input(Submit('submit', 'Submit'))

class Meta:
    model = Persona
    fields = ('nome', 'cognome', 'ruolo')

```

```

from crispy_forms.helper import FormHelper

regola di massima: se il form va modificato dopo l'istanziazione, si assegna l'helper a var di istanza, altrimenti di classe

FormHelper offre moltissimi attributi (lista completa in crispy\_forms/helper.py) per customizzare il form. Nell'esempio usiamo
istanza.form_id: ID assegnato al tag <form>
istanza.form_method: metodo assegnato all'attributo method di <form> (def POST)
istanza.add_input(): metodo per aggiungere input_object al form, nel nostro caso un'istanza di Submit che genera un pulsante submit
Submit('submit', 'Submit'): valori assegnati agli attributi name e value dell'elemento <input>
Submit provvede inoltre a popolare gli attributi dell'input con altri valori (es. classi bootstrap) - se non stiamo usando uni_form, assegna le classi btn btn-primary
la tag {% crispy %} accetta due parametri:
    nome assegnato all'oggetto form nel context (def form)
    attributo helper di form → assegnando il nome helper all'istanza di FormHelper sarà sufficiente usare {% crispy form %} invece di
    {% crispy form form.helper %}

l'app offre lo strumento Layout che permette di modificare radicalmente il modo in cui i campi vengono renderizzati (stabilire l'ordine dei campi, inserirli in <div> o altre strutture, aggiungere HTML, id, classi, attributi,...)

per usare la classe Layout è sufficiente assegnarla all'attributo layout di un'istanza di FormHelper → l'oggetto viene popolato da layout objects che si possono identificare come componenti del form
accetta un numero arbitrario di oggetti

# ...
helper.layout = Layout(
    Div(
        HTML("<p>Inserisci i dati del socio: </p>"),
        Field('nome', style="color: red;", cass_class="bg-dark", title='Nome'),
        Field('cognome', style="color: orange;", cass_class="bg-dark", title='Cognome'),
        Field('ruolo', style="color: green;", cass_class="bg-dark", title='Ruolo'),
        css_class="d-flex justify-content-between"
    ),
    Submit('submit', 'Submit')
)

tutte le classi utilizzate vengono importate da crispy_forms.layout
approfondisci i layout

```

## Django Users

strumenti auth a disposizione di default per gestire gli utenti ed i gruppi

le troviamo in settings.py sia tra le app che tra i middleware → di default si crea la tabella User nel file db dopo la prima migrazione

unico utente (per ora): l'amministratore

per 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 restituisce "Anonymous" se non è loggato, "admin" se eseguiamo il login da .../admin/

```

### Creazione di utenti

un admin lo fa dal pannello di controllo  
un utente anonimo può registrarsi tramite FBV e CBV

La sezione contiene il codice dell'esercitazione **biblio3**

creiamo l'ultima versione di biblioteca, con app gestione  
struttura dei modelli: libri associati a N copie, ciascuna può essere in prestito oppure no, con il prestito operato solo da utenti registrati  
cambia la tabella Copia: rimuoviamo la logica del prestito scaduto (rimane solo la data dell'ultimo prestito) e aggiungiamo una fk a user

**funzionalità esercitazione: (logseq)**

**navigabilità**

(screen su logseq)

```

# gestione/models.py

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

# Create your models here.

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(disp) + " disponibili"
        return out

    class Meta:
        verbose_name_plural = "Libri"

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"

cominciamo con
comando iniziale di popolamento del db libri/copie
home di biblio con navbar e footer per raggiungere facilmente le nostre funzionalità
home di gestione per interagire con le copie (prima va implementata l'autenticazione)

```

(screen su logseq)

### Registrazione utente

meccanismi già quasi completamente implementati da django  
saremo noi dopo a imporre vincoli aggiuntivi sulle app

```
# urls.py

from django.contrib.auth import views as auth_views
urlpatterns = [
    path('admin/', admin.site.urls),
    re_path(r'^$|^(?P<home>home)$', 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"),
```

per la registrazione usiamo una createview che usa il modelform specifico UserCreationForm

```
# views.py

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")
```

poiché questo modelform non usa crispy helper, dobbiamo farlo manualmente, inserendo anche l'elemento submit

```
...
{% block content %}
<form method="post"> {% csrf_token %}
{{form | crispy}}
<input type="submit" class="btn btn-success" value="Registrati ora!">
</form>
{% endblock %}
```

per il login, non dobbiamo aggiungere nessuna logica, soltanto fornire il template con cui renderare il form → poiché non ho scritto la view (la prendo da django), seguo le regole di default per template e success url:

template: deve chiamarsi login.html e trovarsi in biblio3/templates/registration  
success url: hardcoded in settings.py come LOGIN\_REDIRECT\_URL  
usiamo questo per distinguere quando l'utente approda alla home come utente registrato o come anonimo:

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

per il logout è uguale, chiamando il template biblio3/templates/registration/logged\_out.html

### Funzionalità per utenti loggati

come impedire agli utenti anonimi di accedervi:

1. "nascondere" gli url da template → **non va bene! posso comunque scrivere gli url a mano!**

2. accedere tramite le view a user.is\_authenticated: va, **ma devo riscrivere boilerplate per ogni view da proteggere con login!**

### 3. django auth decorators

(soluzione ottimale) → permette di fare pre-processing su ogni view da proteggere, tramite un semplice decoratore

```
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)
```

se l'utente non è loggato viene redirectato verso LOGIN\_URL (in settings.py) → nel nostro caso aggiungiamo un parametro che ci permette di distinguere quando un utente approda sul login in autonomia oppure in seguito al tentativo di accesso ad una pagina protetta

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

```
{% 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 %}
```

### 4. access mixin

per le CBD **non posso usare decorators** (posso, ma scomodo e inefficiente) → sfrutto l'ereditarietà multipla, in particolare con la classe LoginRequiredMixin, che aggiunge in automatico il decoratore ai metodi fondamentali ereditati dalla CBV in questione

gli access mixin devono essere i primi ad essere ereditati! l'ordine di eredità non è indifferente!

### Gruppi

abbiamo diverse tipologie di utenti nella nostra app → per diversificare i permessi sfruttiamo il concetto dei **gruppi**, i quali in django sono proprio contenitori di permessi

gli utenti possono essere associati a 0+ gruppi, **"ereditandone" i permessi**

#### permessi

app | risorsa | azione nella forma generica  
possiamo tramite codice aumentarne la granularità a livello di attributo e tabella  
precissimo, e molto poco scalabile → per questo i gruppi

#### creazione di gruppi

da codice

da admin panel

#### aggiunta utenti ai gruppi

evitiamo di farlo manualmente da admin panel, facciamo in modo che avvenga in automatico alla registrazione

```
# views.py
```

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

```
# 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

```

ovvero ovverrido il metodo `save`, chiamando il padre, recuperando il gruppo che mi interessa e aggiungendovi l'utente, e ritornando il valore che avrebbe ritornato il padre

avviene similmente per i bibliotecari

poiché solo un admin può iscrivere un bibliotecario, possiamo creare una `CreateUserView` protetta con `access mixin permission required` → specificando la variabile `permission_required` possiamo fornire la condizione necessaria per accedere alla view, in questo caso usiamo "is\_staff"

usiamo queste condizioni nei template per menu personalizzati in funzione della tipologia di utente

```

<!-- gestione/templates/gestione/home.html -->
<!-- ... -->

{%- 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:situazioneb' %}" 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 %}

<!-- ... -->

```

#### protezione views con criterio di appartenenza ai gruppi HTML

possiamo usare decoratori appositi per verificare le autorizzazioni

```

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):
    ...

```

#### django-braces HTML

django non offre molto come mixins per gruppi - si installa la lib `django-braces` che offre molto codice per risolvere problemi ricorrenti

alcuni mixin: `GroupRequiredMixin`, `SuperUserRequiredMixin`, `MultiplePermissionRequiredMixin`, ...

`pip install django-braces`

[django-braces docs](#)

osservazioni:

per passare un test di appartenenza **occorre essere loggati**

l'admin (in quanto `is_superuser`) passa sempre tutti i test pur non essendo esplicitamente iscritto ai gruppi  
il redirect in seguito ad operazioni non consentite dipende dal modo in cui abbiamo protetto le view:

- Esempio: loggato come lettore
  - CASO A: Provo ad andare in `localhost:8000/registrob/#solo per admin: registrare bibliotecario`
  - CASO B: Provo ad andare in `localhost:8000/gestione/situazioneb/#per bibliotecari: #situazione prestiti di tutti i libri`

