

Models in Django

Approfondimenti

I modelli in Django

Abbiamo dato una rapida occhiata al concetto di Model in django.

E' un concetto fondamentale del design pattern MVT, ovverosia **Model View Template**.

E' arrivato il momento di approfondire meglio il concetto di Model

Filosofia ORM

Object Relational Mapping

Django usa ORM per la gestione di dati strutturati.

Possiede librerie interne in grado di “tradurre” oggetti (istanze di classi) in entità fruibili come dati in un database.

Le classi stesse rappresentano le tabelle dei DB

I loro oggetti istanziati rappresentano le entry di queste tabelle

Il processo di traduzione tra codice Python e comandi\query inerenti al DB sottostante si chiama **migrazione**

Peculiarità

L'approccio ORM di django è molto comodo per lo sviluppatore:

- Possibilità di creare ed usare database complessi senza dover scrivere query
- Le classi ORM sono ben integrate con il resto del nostro codice python\DTL
- Ci rendiamo “slegati” dalle effettive scelte implementative del nostro DB
- Possiamo sfruttare concetti legati all’ereditarietà di classi per massimizzare il riutilizzo dei nostri modelli
- Possiamo personalizzare a piacere il comportamento delle classi modello con l’aggiunta di metodi e META-informazioni.
- Il sistema delle migrazioni, come vedremo, ci permette di avere uno strumento aggiuntivo per evitare “danni” accidentali al nostro DB
- Funzioni comunemente usate nei DB, import\export\CRUD+L etc... sono praticamente “gratis” tramite il **django View system**

Esempio

Semplice gestione di una biblioteca

Si comincia con un nuovo progetto, chiamiamolo “biblio”.

Diversamente dalle altre volte, ora creeremo una app per questo progetto.

L'app “gestione”, che appunto gestisce un semplice DB di libri

Occorrerà poi quindi definire gli attributi del nostro database...

Un aiutino



Creazione della directory del progetto

In quella cartella si crei un *venv*: `pipenv install django`

Si entri nel *venv*: `pipenv shell`

Si crei il progetto: `django-admin startproject biblio`

Si crei l'app: `python manage.py startapp gestione`

First things first



Creiamo le cartelle:

templates nella root e templates/gestione nell'app

Mettiamo il nostro base.html in biblio/templates

Assicuriamoci che i template siano in grado di essere trovati, in settings.py:

```
#attenzione: "os" richiede "import os"
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, "templates")],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

L'app gestione



Deve essere installata, in settings.py

```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'gestione'
]
```


gestione: urls.py

Tipicamente non viene creato.

Lo si crei, facendo copia ed incolla da urls.py di biblio

Si aggiunga una view di benvenuto e l'attributo "app_name"

In biblio/urls.py, invece

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('gestione/', include('gestione.urls'))
]
```

Creazione del DB

Partiamo quindi con una cosa semplice

Una sola tabella

Un libro è caratterizzato da Titolo, Autore, Pagine, e da una data dell'ultimo prestito.

Quindi, semplicemente, una copia per libro.

E per il momento non abbiamo utenti che restituiscono\prendono libri

In 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)
```

Apply migration

Preparare la migrazione “app-level”

```
python .\manage.py makemigrations gestione
Migrations for 'gestione':
  gestione\migrations\0001_initial.py
    - Create model Libro
```

Rendere effettiva la migrazione:

```
python .\manage.py migrate
```

Apply migration

Preparare la migrazione “app-level”

```
python .\manage.py makemigrations gestione  
Migrations for 'gestione':
```

```
gestione\migrations\0001_initial.py
```

```
- Create model Libro
```

Rendere effettiva la migrazione:

```
python .\manage.py migrate
```

```
# Generated by Django 4.0.2 on 2022-02-15 08:59
```

```
from django.db import migrations, models
```

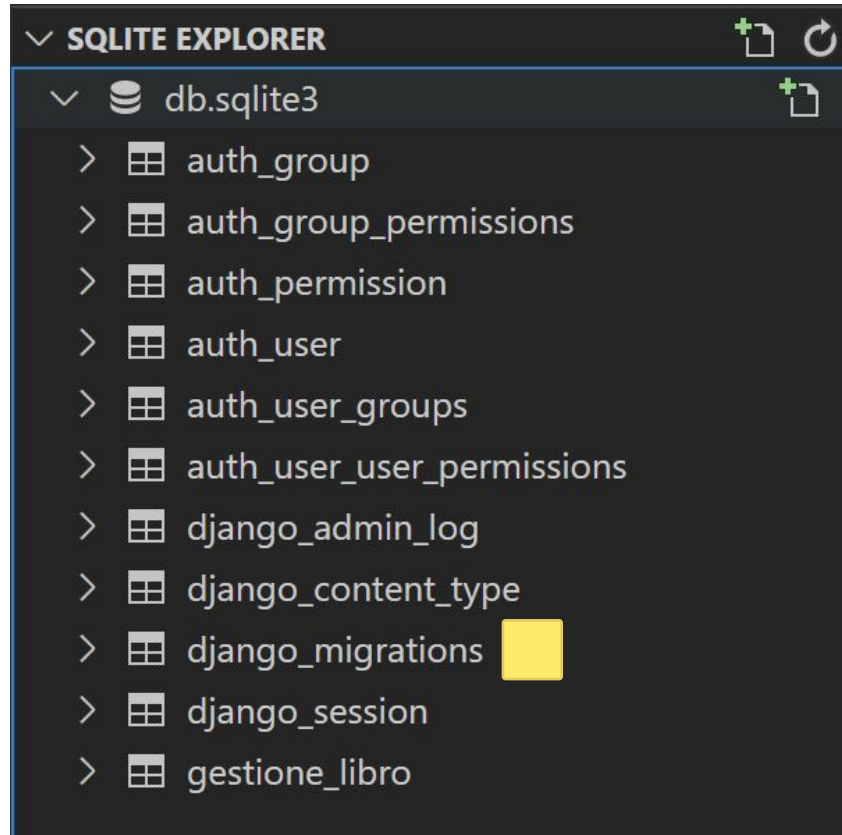
```
class Migration(migrations.Migration):
```

```
    initial = True
```

```
    dependencies = [  
    ]
```

```
    operations = [  
        migrations.CreateModel(  
            name='Libro',  
            fields=[  
                ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),  
                ('titolo', models.CharField(max_length=200)),  
                ('autore', models.CharField(max_length=50)),  
                ('pagine', models.IntegerField(default=100)),  
                ('data_prestito', models.DateField(default=None)),  
            ],  
        ),  
    ]
```

Quali (altri) database?



Inserire elementi nel database

Diversi modi:

- Programmaticamente
- Tramite console dell'admin
- Tramite Views and function views



Inserire elementi nel database

Diversi modi:

- Programmaticamente
 - Ossia usando codice python; tipicamente si usa nel setup iniziale, quando si importano\esportano dati, routine di manutenzione e controllo etc...
- Tramite console dell'admin
 - Permette di fare tutto quello elencato nel punto precedente tramite una comoda interfaccia grafica web-based
- Tramite Views and function views
 - Modifiche ai dati del DB scatenate da richieste client-side

Programmaticamente

python shell con: *python manage.py shell*

```
>>> from gestione.models import Libro
>>> q = Libro.objects.all()
>>> print(str(q)+" tipo di dato "+str(type(q)))
<QuerySet []> tipo di dato <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.pagine
100
>>> from django.utils import timezone
>>> l.data_prestito = timezone.now()
>>> print(l)
Promessi Sposi di Alessandro Manzoni in prestito dal 2022-02-15 09:31:06.183132+00:00
>>> l.save()
>>> Libro.objects.all()
<QuerySet [<Libro: Promessi Sposi di Alessandro Manzoni in prestito dal 2022-02-15>]>
>>> lq = Libro.objects.get(pk=1)
>>> lq
<Libro: Promessi Sposi di Alessandro Manzoni in prestito dal 2022-02-15>
```

Aggiungere tanti libri

Perchè non fare una funzione apposita?



```

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 inizzializzo
    | #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 k in libridict:
    | #     print(str(libridict[k]))
    |
    | 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]
    |         else:
    |             l.data_prestito = libridict[k][i]
    |     #print(l)
    |     l.save()
    |
    | print("DUMP DB")
    | print(Libro.objects.all()) #controlliamo

```

Ok, ma dove...?

Queste funzioni possiamo definirle come “operazioni di amministrazione”, oppure “one-shot operations”, da eseguirsi, per esempio, una sola volta in seguito allo start-up del webserver django.

Manualmente: possiamo usare uno script .py... ma ci dobbiamo ricordare di eseguirlo.

Automaticamente:

come posso dire a django “quando il server è avviato, esegui queste operazioni di amministrazione?”

Django one-shot operations

Possono essere messe in <root_prj>/urls.py

```
from django.contrib import admin
from django.urls import path, include
from .initcmds import * #definizione di erase e init_db

urlpatterns = [
    path('admin/', admin.site.urls),
    path('gestione/', include('gestione.urls'))
]

erase_db()
init_db()
```

Django custom commands

<https://docs.djangoproject.com/en/4.0/howto/custom-management-commands/>

Come è il DB ora?

SQL ▼



1

/ 1



1 - 5 of 5

id	titolo	autore	pagine	data_prestito
12	Promessi Sposi	Alessandro Manzoni	832	2022-02-15
13	1984	George Orwell	328	2022-02-14
14	Odissea	Omero	414	2022-01-15
15	La Fattoria degli Animali	George Orwell	141	2022-01-14
16	Iliade	Omero	263	2021-02-15

Gestione del DB tramite admin

Andando su <http://127.0.0.1:8000/admin/>

Possiamo fare tutto quello che abbiamo visto tramite le funzioni

Ovviamente però **deve** esistere un superutente admin

python manage.py createsuperuser

ed in gestione/admin.py

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

Models

Function Views ed integrazione con DTL

Accesso al DB da parte dell'utente

Fino ad ora abbiamo visto operazioni di amministrazione.

Come tali, debbono essere svolte da superutenti, amministratori, o generici membri “staff” del sito.

Non possono e non devono ovviamente essere esposte al “pubblico”

Response function views

Per ora sappiamo rispondere a richieste GET su HTTP che si attivano nel momento in cui l'utente cerca risorse specifiche all'interno dell'albero di risorse url dinamicamente generate.

Ma abbiamo anche visto come “giocare” con il DB tramite normalissimo codice python.

E' tempo di unire le due cose

Lista dei libri

Creiamo una view, con tanto di template in grado di mostrare il contenuto del DB.

In gestione/views.py:

```
from django.http import HttpResponse
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)
```

Il Template (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 %}
        </ul>
    {% else %}
        <p>Non ci sono libri!</p>
    {% endif %}

    </center>

{% endblock %}
```

L'url corrispondente in gestione/urls.py

```
from django.urls import path
from .views import *

app_name = "gestione"

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

Lista di Libri

Ci sono ben 5 libri in questa biblioteca!

-
-
-
-
-

Promessi Sposi di Alessandro Manzoni in prestito dal 2022-02-15
1984 di George Orwell in prestito dal 2022-02-14
Odissea di Omero in prestito dal 2022-01-15
La Fattoria degli Animali di George Orwell in prestito dal 2022-01-14
Iliade di Omero in prestito dal 2021-02-15

Esempio

Isoliamo i libri “lunghi”.

Un libro “lungo”, a.k.a. “mattone” è tale se supera le 300 pagine.

Si faccia una function view che riutilizzando lo stesso template di prima permetta di visualizzare solo i libri mattone.

Occorre operare un'operazione di filtro sul QuerySet del nostro Model. La lista risultante verrà quindi passata tramite il dizionario di contesto

La view “filtrante”

In 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, template_name=templ, context=ctx)
```

QuerySet.filter(...)

I filtri possono essere arbitrariamente complessi e vanno oltre il concetto di uguaglianza. Nel caso appena visto infatti non vogliamo i libri il cui attributo pagine sia uguale ad un certo numero, bensì filtriamao\escludiamo ad una certa soglia.

Più genericamente:

<nome_attributo>__[gt,lt,gte,lte,] = value,


Altre operazioni prevedono l'ordinamento, equivalente alla clausola “orderBy”

<https://docs.djangoproject.com/en/4.0/ref/models/queriesets/>

Altre operazioni sui DB tramite i metodi accessori di Django

```
>>> from gestione.models import Libro
>>> qs = Libro.objects.all()
>>> qs.order_by('data_prestito') #Ordina in senso crescente secondo il criterio "data_prestito"
<QuerySet [<Libro: Iliade di Omero in prestito dal 2021-02-15>, <Libro: La Fattoria degli Animali di George
Orwell in prestito dal 2022-01-14>, <Libro: Odissea di Omero in prestito dal 2022-01-15>, <Libro: 1984 di
George Orwell in prestito dal 2022-02-14>, <Libro: Promessi Sposi di Alessandro Manzoni in prestito dal
2022-02-15>]>
qs.order_by('-data_prestito') #come sopra ma in ordine decrescente
<QuerySet [<Libro: Promessi Sposi di Alessandro Manzoni in prestito dal 2022-02-15>, <Libro: 1984 di
George Orwell in prestito dal 2022-02-14>, <Libro: Odissea di Omero in prestito dal 2022-01-15>, <Libro: La
Fattoria degli Animali di George Orwell in prestito dal 2022-01-14>, <Libro: Iliade di Omero in prestito dal
2021-02-15>]>
>>> a = qs.order_by("-pagine")[:2] #prendi i due libri con più pagine.
>>> for l in a:
...     print(str(l) + " con pagine " + str(l.pagine))
...
Promessi Sposi di Alessandro Manzoni in prestito dal 2022-02-15 con pagine 832
Odissea di Omero in prestito dal 2022-01-15 con pagine 414
```

Altre operazioni sui DB tramite i metodi accessori di Django

```
>>> qs0 = Libro.objects.filter(autore__iexact="Omero") #Case insensitive
>>> qs1 = Libro.objects.filter(autore__iexact="George Orwell")
>>> qs0
<QuerySet [<Libro: Odissea di Omero in prestito dal 2022-01-15>, <Libro: Iliade di Omero in prestito dal 2021-02-15>]>
>>> qs1
<QuerySet [<Libro: 1984 di George Orwell in prestito dal 2022-02-14>, <Libro: La Fattoria degli Animali di George Orwell in prestito dal 2022-01-14>]>
>>> q01 = qs0 | qs1 #Unione dei query set 
>>> q01
<QuerySet [<Libro: 1984 di George Orwell in prestito dal 2022-02-14>, <Libro: Odissea di Omero in prestito dal 2022-01-15>, <Libro: La Fattoria degli Animali di George Orwell in prestito dal 2022-01-14>, <Libro: Iliade di Omero in prestito dal 2021-02-15>]>
```

E se non bastasse

Il metodo filter e la sua versatilità nel definire criteri di selezione dovrebbe bastare per qualsiasi cosa ci venga in mente.

Qualora non fosse così, si possono comunque utilizzare *raw queries* in caso siamo sicuri che il DB sottostante sia di fatto un **database** 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)
```



Attenzione:

Mentre `all()/exclude()` e `filter()` restituivano oggetti di tipo `QuerySet`, una raw query come quella sopra restituisce un RawQuerySet. E' un oggetto sostanzialmente diverso e non ha il metodo "count" che abbiamo usato nel template in html/DTL.



Esercizi [1]

- Scrivere una function view di risposta ad un url che tramite metodo GET prende in ingresso il parametro “autore” e mostri nell’html i libri scritti dall’autore passato come parametro
- Come sopra, ma il parametro dev’essere passato tramite url path
e.g. `path(“autore/<str:autore>/”, autore_path, nome=“autorepath”)`

Aggiunta di entries nel DB

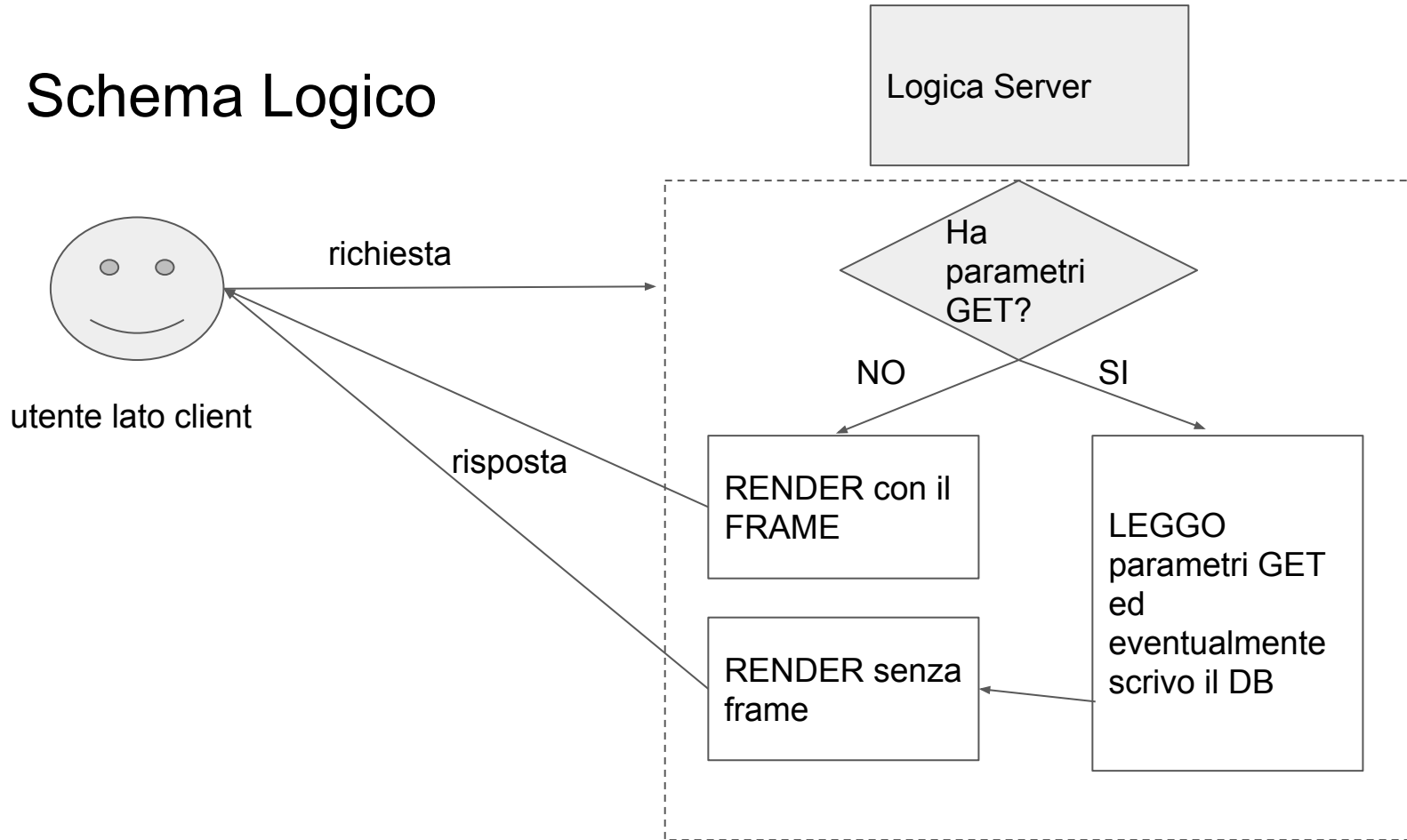
Potremmo voler dare all'utente la possibilità di **scrivere** nel DB e non solo di leggere come abbiamo visto negli esempi precedenti

L'idea è la seguente: scrivere un html form in grado di prendere tramite textfield gli attributi che ci interessano del libro da creare, quindi **titolo, autore e pagine**.

Il form ha un pulsante: se premuto ci riporterà nella stessa pagina ma con i parametri GET impostati tramite url.

La rispettiva view function dovrà quindi discriminare il caso in cui l'utente "arriva" alla pagina per la prima volta (e quindi dovrà comparire il form) o nel caso in cui l'utente abbia già riempito il form e il libro dovrà essere creato con gli attributi precedentemente inseriti

Schema Logico



La view function

```
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/crealibro.html",
                  context={"title": "Crea Autore", "message": message})
```

```
path("crealibro/", crea_libro, name="crealibro").
```

Il template

```
gestione > templates > gestione > crealibro.html
1  {% extends "base.html" %}
2
3  {% block head %} {% endblock %}
4
5  {% block title %}{{title}}{% endblock %}
6
7  {% block content %}
8
9      <center>
10     <h1>{{title}}</h1>
11
12     {% if message == "" %}
13
14     <form action="/gestione/crealibro">
15         <label for="autore">Autore:</label><br>
16         <input type="text" id="autore" name="autore" ><br>
17         <label for="titolo">Titolo:</label><br>
18         <input type="text" id="titolo" name="titolo" ><br>
19         <label for="pagine">Pagine:</label><br>
20         <input type="text" id="pagine" name="pagine" ><br><br>
21         <input type="submit" value="Crea Entry">
22     </form>
23
24     {% else %}
25
26     <p> {{message}} </p>
27     <a href="{% url 'gestione:listalibri' %}">Lista Libri</a>
28
29     {% endif %}
30
31 </center>
32
33 {% endblock %}
```

Ecco come il template si “accorge” se è la prima o la seconda visita

Alla pressione del pulsante, torno su questa pagina. Ma con i parametri GET impostati secondo gli “id” dei textfield del form

app namespace e alias per la risoluzione di un link

http://127.0.0.1:8000/gestione/crealibro

Crea Autore

Autore:

Autore Fittizio

Titolo:

Libro Fittizio

Pagine:

ciao!

Crea Entry

http://127.0.0.1:8000/gestione/crealibro/?autore=Autore+Fittizio&titolo=Libro+Fittizio&pagine=ciao%21



Crea Autore

Creazione libro riuscita! Pagine non valide. Inserite pagine di default.

[Lista Libri](#)

Lista di Libri

Ci sono ben 6 libri in questa biblioteca!

Promessi Sposi di Alessandro Manzoni in prestito dal 2022-02-15
1984 di George Orwell in prestito dal 2022-02-14
Odissea di Omero in prestito dal 2022-01-15
La Fattoria degli Animali di George Orwell in prestito dal 2022-01-14
Iliade di Omero in prestito dal 2021-02-15
Libro Fittizio di Autore Fittizio in prestito dal 2022-02-15

http://127.0.0.1:8000/gestione/listalibri

Change libro

Libro Fittizio di Autore Fittizio in prestito dal 2022-02-15

Titolo:	Libro Fittizio
Autore:	Autore Fittizio
Pagine:	100
Data prestito:	2022-02-15 Today

Note: You are 1 hour ahead of server time.

Esercizi [2]

- La view vista in precedenza crea una nuova entry.
 - E se volessi fare una view in grado di modificare una entry esistente?
 - E se volessi fare una view in grado di cancellare una entry esistente?
- Suggerimenti nella prossima slides

Modifica

Quale libro?
Passato tramite url path

127.0.0.1:8000/gestione/modificalibro/Promessi%20Sposi/Alessandro%20Manzoni/

Modifica Libro

Modifica il libro: Autore:

Alessandro Manzoni

Titolo:

Promessi Sposi

Pagine:

832

Modifica

Campi di testo editabili,
partendo dai valori
correntemente salvati nel DB

Modifica, queries e 404

```
def modifica_libro(request,titolo,autore):  
    libro = get_object_or_404(Libro, autore=autore, titolo=titolo)  
    [...]
```

get_object_or_404 fa una query sulla tabella Libro con clausole di selezione su autore e titolo.

Se la query restituisse più di un risultato: **Exception**

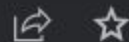
Se la query non restituisse nessun risultato: redirectione su pagina 404 (res not found)

Richiede **from *django.shortcuts* import *get_object_or_404***

Cancellazione del record di un libro

Come lo selezioniamo?

127.0.0.1:8000/gestione/cancellalibro/



Elimina Libro

Scegli un libro da cancellare: 12: Alessandro Manzoni Promessi Sposi ▼

Elimina

HTML select multiple attributes: https://www.w3schools.com/tags/att_select_multiple.asp

Cancellazione del record di un libro

Come lo cancelliamo?

127.0.0.1:8000/gestione/cancellalibro/



Elimina Libro

Scegli un libro da cancellare: 12: Alessandro Manzoni Promessi Sposi ▼

Elimina

<pkid>: *<Autore>* *<Titolo>* Mi basta manipolare questa stringa per ottenere la primary key value dell'elemento selezionato. In questo modo `Libro.objects.get(pk=pkid)`

Discussione

Siamo quindi in grado di accedere in lettura e scrittura ad un DB anche tramite azioni dell'utente. In particolare abbiamo visto ed implementato tutti i metodi **CRUD** (create, read\retrieve, update and delete).

Ma ci sono ancora alcuni problemi...

DRY: completamente violato

Molte view si somigliano.

Così come molto codice HTML\DTL.

Per risolvere gli esercizi c'è stato molto “copia ed incolla”

Il codice è tutto fuorché elegante...

Soluzione: a breve introdurremo le **Django Class Views and Forms**

DB Access control

Ma vogliamo proprio lasciare accesso a chiunque in lettura\scrittura a tutte le mie tabelle del mio DB?

E se volessi introdurre dei vincoli aggiuntivi sui valori assegnabili agli attributi delle mie entry?

A breve introdurremo anche **Django Access control Mixin and Permission**