

**django**

startapp


# startapp

Come abbiamo già detto, un'app è un'applicazione web che ha funzionalità specifiche e definite. Per creare una nuova app e integrarla in un progetto è necessario seguire una ben precisa procedura.

Per creare una nuova app si utilizza il comando **startapp** <nome\_app>.

All'interno della cartella del progetto, esegui:  
`python manage.py startapp soci`

django genera in automatico una cartella  
contenente una serie di file

A green arrow points from the text box on the left to the file structure on the right.

```
soci/  
    __init__.py  
    admin.py  
    apps.py  
    migrations/  
        __init__.py  
    models.py  
    tests.py  
    views.py
```



# startapp

## A cosa servono questi file?

**admin.py** → qua si registrano i *model* (tabelle del database) da includere nella sezione *admin* di django;

**apps.py** → include le principali configurazioni dell'applicazione;

**models.py** → *model* dell'applicazione;

**tests.py** → qui si aggiungono test per l'applicazione;

**views.py** → le *view* specifiche dell'applicazione.

**migrations** → questa cartella contiene le migrazioni di database dell'applicazione.\*

\* per maggiori informazioni sulle migrazioni,  
consultare la sezione *model*.




# startapp

Non essendo strettamente necessari, il comando *startapp* non genera una cartella *templates* e un file *urls.py*, questo perché non tutte le app devono necessariamente servire dei template o utilizzare degli url. Quando però sono necessari, è opportuno creare la cartella *templates* e il file *urls.py* all'interno dell'app per rendere il codice meglio organizzato e più facilmente fruibile.

Crea la cartella *templates* all'interno dell'app 'soci'.

Una buona pratica per evitare conflitti con altri template è quella di creare, all'interno della cartella *templates* interna ad un'app, una sotto cartella riportante il nome dell'app, in questo modo:

```
app_name/  
|   templates/  
|   |   app_name/  
|   |   |   template.html
```



Questo può essere molto utile soprattutto considerando che django, nel momento in cui deve servire un template, passa in rassegna tutte le cartelle con il nome corrispondente alla stringa assegnata alla chiave *DIRS* in *settings.TEMPLATES* all'interno del progetto finché non trova il template corrispondente.

# startapp

Per fare in modo che django tenga traccia della nostra applicazione, è necessario “registrarla”.  
In settings.py > INSTALLED\_APPS aggiungi '**soci.apps.SociConfig**':

Una volta aggiunta una nuova app al progetto, è necessario adoperare i comandi **makemigrations/migrate** per fare sì che vengano create le tabelle sul database. \*

\* per maggiori informazioni sulle migrazioni, consultare la sezione *model*.

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

# startapp

Crea il file `urls.py` all'interno dell'app 'soci'.

Il file `urls.py` creato in un'app viene trattato alla stessa maniera di uno presente nel progetto, ma è buona pratica assegnare alla variabile `app_name` il nome dell'app. Questa operazione crea un *namespace* specifico per il file ed evita conflitti con altri url utilizzati nei template che potrebbero avere lo stesso nome:

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

app_name = 'soci'
```

Questa è la sintassi per assegnare un url django ad un attributo di un elemento HTML (es. href):

```
{% url 'path' %}
```

Utilizzando `app_name` la sintassi diventerà la seguente:

```
{% url '<app_name>:path' %}
```

Per permettere a django di trovare gli url di un'app è necessario utilizzare nel file `urls.py` del progetto la funzione `include` nel seguente modo:

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

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

models



# models

Il *model* è lo strumento utilizzato da django per strutturare i dati sul database. Contiene le informazioni sui campi e i comportamenti dei dati memorizzati. Solitamente, ogni *model* viene mappato su una singola tabella del database.

- Ogni *model* è una classe Python che eredita da **django.db.models.Model**;
- Ogni attributo del model rappresenta un campo del database;
- django genera automaticamente delle API di accesso al database

```
from django.db import models

class Auto(models.Model):
    modello = models.CharField(max_length=20)
    casa = models.ForeignKey(CasaProduttrice)
    anno = models.DateField()
    cc = models.IntegerField()
```

Le classi assegnate agli attributi fanno parte del modulo `models` e definiscono il tipo di dato che verrà salvato nel campo specifico.



# models

Nel file `models.py` all'interno dell'app `soci`, inserisci il codice che segue:

```
from django.db import models

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

**nome** e **cognome** sono campi del model. Ogni campo è specificato come attributo di classe e ogni attributo è mappato ad una colonna del database.

- il nome della tabella, **soci\_persona**, viene generato automaticamente da django ma può essere sovrascritto;
- un campo **id** viene aggiunto automaticamente ma, anche questo, può essere modificato. Questo è la **primary-key** del model;
- a fianco è stata usata la sintassi di PostgreSQL, ma django può utilizzare anche SQLite(default), MySQL e Oracle.

Il modello **Persona** creerà una tabella sul database con il seguente codice:

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

# models

Le migrazioni sono il modo in cui django estende le modifiche fatte sui model al database. Ci sono diversi comandi che si utilizzano per fare migrazioni. I più comuni sono **makemigrations** e **migrate**.

Si può pensare alle migrazioni come ad un *version control system* (sistema che rileva le modifiche su uno o più file) dello schema del database.

**makemigrations** → crea nuove migrazioni basate sui cambiamenti rilevati nei model. Eseguendo il comando, viene automaticamente generato un file nel quale vengono elencate la creazione di nuovi model o le modifiche a model già esistenti.

**migrate** → è il comando che applica le migrazioni.

Una volta eseguite le migrazioni e generato il file relativo con **makemigrations**, **migrate** esegue i comandi presenti nel file generato e applica le modifiche al database.



# models

Da terminale, esegui:  
`python manage.py makemigrations soci`

In questo modo, in `soci/migrations` verrà generato il file `0001_initial.py`. Questo conterrà del codice Python traducibile in:

```
(tutorial)$ python manage.py sqlmigrate soci 0001
BEGIN;
--
-- Create model Persona
--
CREATE TABLE "soci_persona" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "nome" varchar(50) NOT NULL, "cognome" varchar(50) NOT NULL);
COMMIT;
```

Ora esegui:  
`python manage.py migrate`

Sul nostro database, tra le altre (innumerevoli!) tabelle autogenerate da django, troveremo anche la tabella **soci\_persona**.



# models

django offre anche una serie di tipi di campi che rappresentano relazioni tra tabelle:

- ForeignKey
- ManyToManyField
- OneToOneField

**ForeignKey** rappresenta una relazione *many-to-one* ( $n$  a 1).

Richiede due *argument* posizionali:

- la classe (model) al quale il model si relaziona;
- il parametro `on_delete` che accetta, tra i valori più usati:
  - `models.CASCADE` → eliminazione a cascata. django emula il comportamento del vincolo SQL "ON DELETE CASCADE";
  - `models.PROTECT` → previene l'eliminazione dell'oggetto con il quale si crea la relazione.



# models

Un altro parametro molto importante è `related_name`, il nome utilizzato dall'oggetto messo in relazione a quello con la *foreign key* per accedervi.

Nel model `Persona` aggiungi il codice che segue:

```
ruolo = models.ForeignKey(Ruolo, on_delete=models.PROTECT, related_name='persone')
```

Nel file `models.py` all'interno dell'app `soci`, inserisci il codice che segue:

```
class Ruolo(models.Model):  
    titolo = models.CharField(max_length=30)  
  
    class Meta:  
        verbose_name_plural = 'Ruoli'
```

Da un'istanza di `Ruolo`, sarà possibile accedere a una lista contenente tutte le `Persone` aventi un determinato ruolo con la sintassi:

```
istanza_ruolo.persone.all()
```



# models

**ManyToManyField** rappresenta una relazione many-to-many. Richiede un argument posizionale: la classe con la quale il model si relaziona.  
django genera automaticamente una tabella intermedia per rappresentare la relazione (reificazione).

**OneToOneField** rappresenta una relazione one-to-one. Concettualmente è simile a ForeignKey con il parametro `unique=True` (parametro che specifica che il valore del campo deve essere unico), ma l'accesso attraverso il `related_name` ritorna direttamente un oggetto singolo.  
Questo viene usato principalmente come primary key di un model che “estende” un altro model.



**createsuperuser**



# createsuperuser

Una delle caratteristiche più interessanti di Django è l'interfaccia di amministrazione automatica. Legge i metadati dei *model* per fornire un'interfaccia rapida ed efficiente in cui utenti selezionati possono gestire i contenuti dell'applicazione web.

Da terminale, esegui:

```
python manage.py createsuperuser
```

L'output su terminale sarà simile a questo:

```
Username (leave blank to use 'admin'): admin
```

```
Email address: admin@admin.com
```

```
Password: ****
```

```
Password (again): ****
```

```
Superuser created successfully.
```

Ora, utilizzando il comando *runserver*, aprendo il browser all'indirizzo `127.0.0.1:8000/admin`, potremo accedere all'interfaccia di login per admin.

A screenshot of the Django administration login page. It has a blue header with the text 'Django administration'. Below the header, there are two input fields: 'Username:' and 'Password:'. Each field has a small eye icon to toggle visibility. At the bottom, there is a blue 'Log in' button.

Django administration

Username:

Password:

Log in

# createsuperuser

Effettuando il login nella pagina di admin con le credenziali inserite dopo l'utilizzo del comando *createsuperuser*, si accede alla pagina di amministrazione vera e propria. Questa conterrà di default due tabelle sotto la voce AUTHENTICATION AND AUTHORIZATION: **Groups** e **User**.

## Site administration

AUTHENTICATION AND AUTHORIZATION	
Groups	<a href="#">+ Add</a> <a href="#">Change</a>
Users	<a href="#">+ Add</a> <a href="#">Change</a>

I model **Groups** e **User** fanno parte dell'autenticazione di default di django. Groups definisce dei gruppi con specifici privilegi. User contiene gli utenti registrati (almeno uno, il profilo appena creato con *createsuperuser*).



# createsuperuser

Per aggiungere i propri model alla pagina di amministrazione è necessario modificare il file *admin.py* generato automaticamente alla creazione dell'app 'soci':

Aggiungi il seguente codice ad *admin.py*:

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

admin.site.register(Persona)
```

Ricaricando la pagina di amministrazione, comparirà anche il model Persona dell'applicazione 'soci'.

SOCI

Personas

+ Add    Change

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

    class Meta:
        verbose_name_plural = 'Persone'
```

django parte dall'assunto che i nomi dei model siano in lingua inglese perciò aggiunge una 's' in fondo al nome del model. Per evitare questo tipo di comportamento è sufficiente aggiungere l'attributo `verbose_name_plural` ai metadati del model:

**views.py**

# views.py

Le *class-based views* (view basate sulle classi) sono metodi alternativi per implementare view come oggetti Python anziché come funzioni (tra questi, *TemplateView*).

Le classi più utilizzate per mostrare dati sono **DetailView** e **ListView**.

**DetailView:** questa *class-based view* si utilizza per visualizzare i dati relativi ad un'entry di una specifica tabella del database. Di default, ritorna un context con l'oggetto *object*, l'istanza dell'entry.

Nel file *views.py* all'interno dell'app 'soci', aggiungi il seguente codice:

```
from django.views.generic.detail import DetailView

class PersonaDetail(DetailView):

    model = Persona
    template_name = 'soci/persona_detail.html'
```

import di DetailView da  
django.views.generic.detail

La class-based view  
sub-classe DetailView

Assegnamo alla classe gli attributi model e template\_name

# views.py

Essendo `DetailView` utilizzata per visualizzare una sola entry di una tabella di un database, è necessario specificare a quale entry la view si riferisce. Per farlo, si passa un parametro nell'url relativo alla view.

Nel file `urls.py` all'interno dell'app 'soci', aggiungi il seguente codice:

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



Questa è la sintassi che django utilizza per passare parametri attraverso gli url.  
In questo caso stiamo specificando che il tipo di dato sarà un *int* e che rappresenterà la *pk* (*primary key*, nel nostro caso l'*id*) della nostra entry.



# views.py

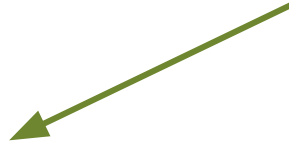
All'interno di templates nell'app 'soci', creiamo il file `persona_detail.html`. Al suo interno utilizzeremo la template syntax di django per mostrare il contenuto della nostra entry.

Crea l'ossatura di HTML all'interno di `persona_detail.html` e aggiungi nel `<body>`:

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

La sintassi “`{{ }}`” si utilizza per inserire nei template delle variabili. *object* è l'istanza dell'entry.

Ora sarà sufficiente aggiungere un'entry nella nostra tabella attraverso l'interfaccia di amministrazione e inserire nel browser l'url `127.0.0.1:8000/soci/persona/1`



Questo può essere un qualsiasi numero che rappresenta un'entry nella tabella sul database.



# views.py

**ListView:** questa *class-based view* ha molte similarità con `DetailView`, ma invece di ritornare una specifica entry di una tabella del database, ritorna **tutte** le entry della tabella. Di default ritorna un context con l'oggetto *object\_list*, la lista di tutte le entry della tabella.

Nel file *views.py* all'interno dell'app 'soci', aggiungi il seguente codice:

```
from django.views.generic.list import ListView  
  
class PersonaList(ListView):  
  
    model = Persona  
    template_name = 'soci/persona_list.html'
```

import di `ListView` da  
`django.views.generic.list`

La class-based view eredita  
sub-classe `ListView`

Assegnamo alla classe gli attributi `model` e `template_name`





# views.py

Ora inseriamo alcune entry nella tabella 'soci' dall'interfaccia di amministrazione, colleghiamo un url alla view **PersonaList** e creiamo, all'interno di `soci/templates/soci/` il file `persona_list.html`.

url da inserire nel file `urls.py`  
all'interno dell'app 'soci'

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

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

Tutte le *template tag* dotate di contenuto (es. `{% block %}`) vengono "chiusure" da tag composte da `{% end<nome_tag> %}`

All'interno di `persona_list.html` inseriamo:

1. Un for loop che itera sull'oggetto `object_list`;
2. Gli attributi di ogni oggetto presente nella lista `object_list`.

# django

[francesco.faenza@unimore.it](mailto:francesco.faenza@unimore.it)