

TUTORIAL DJANGO

Parte 1

Internet, Web e Cloud
Claudia Canali

Impostazioni preliminari

- **Versione di Python : 3.X**

- **Versione di Django**

Per verificare la versione installata

\$ python -m django --version

Una prima applicazione

Applicazione poll

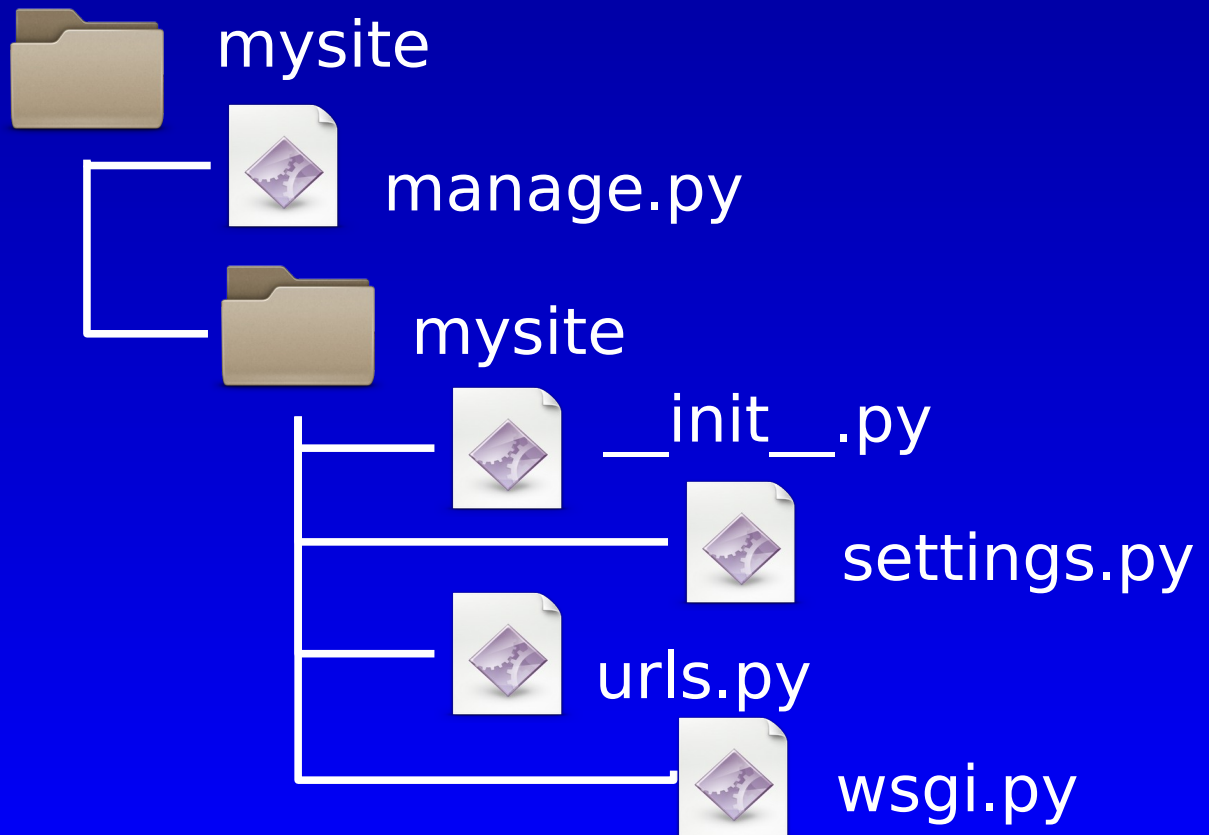
- Semplice **applicazione che gestisce sondaggi (poll)**
- Consiste di **due parti**:
 - Un **sito pubblico** che permette agli utenti di vedere i poll esistenti e di votare
 - Un **sito di amministrazione** che permette di gestire l'applicazione (aggiungere, modificare e cancellare i poll e le risposte)
- L'applicazione farà uso di DB per gestire i dati relativi ai poll
- Iniziamo passo passo...

Primi passi

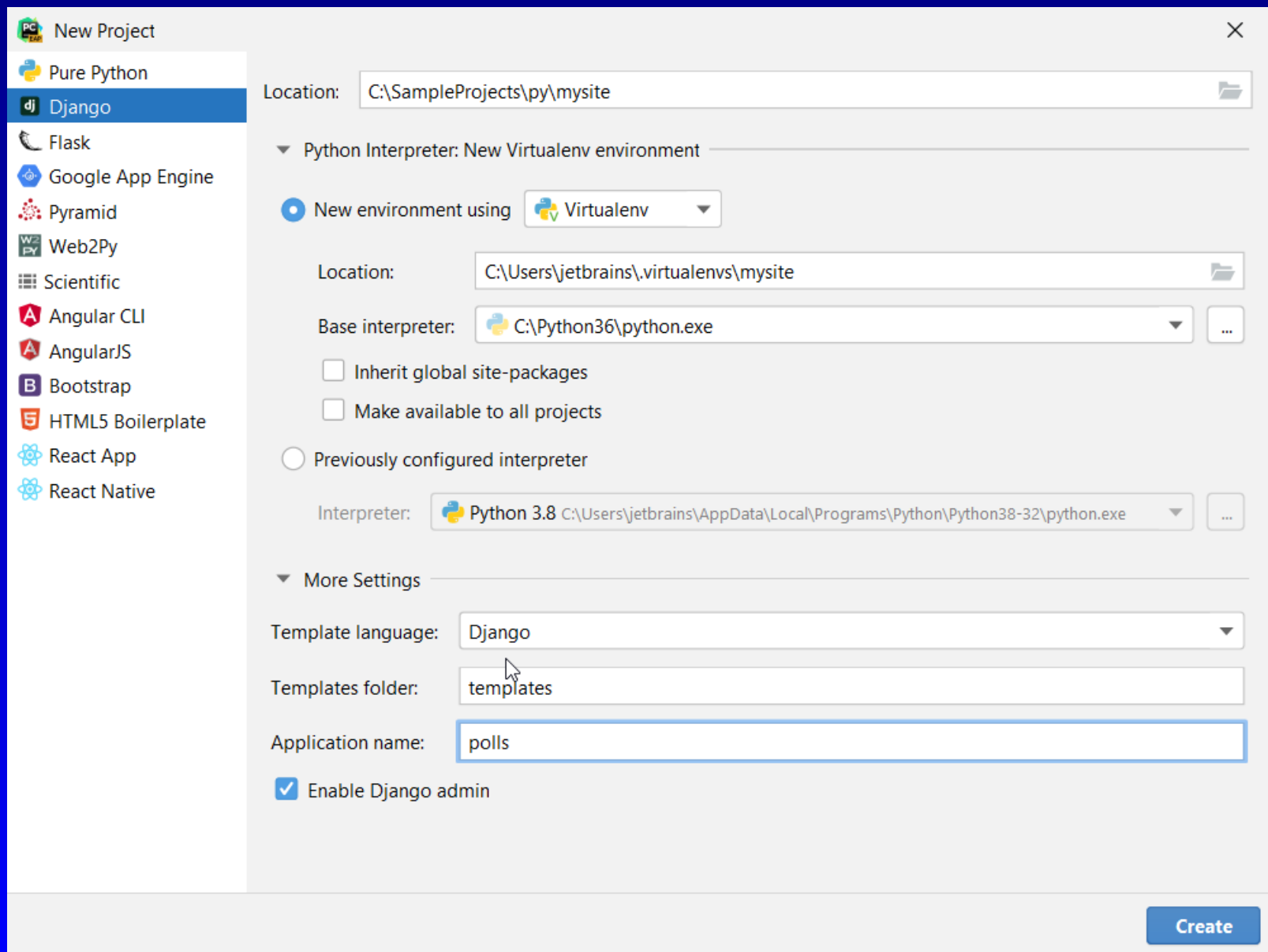
- **Primi step:**
 - 1) Definire la struttura (sito principale e applicazioni)
 - 2) Definire il modello dei dati (struttura DB)
 - 3) Definire gli url per la navigazione del sito
- Creare una **cartella di lavoro**
 - Django, anche se è integrato con un Webserver, **non impone** una location specifica per il codice
 - **Non è obbligatorio mettere** il codice in:
 - /var/www/ o simili
- Si può lavorare direttamente in una qualunque **cartella di lavoro**

Creazione dello scheletro

- Creare la **struttura di base** del progetto mysite (file e cartelle del progetto): utility django-admin.py
 - \$ django-admin startproject mysite



Con Pycharm



The image shows the 'New Project' dialog in PyCharm. On the left is a sidebar with various project templates. 'Django' is selected. The main area is divided into sections for project location, Python interpreter, and Django-specific settings. The project location is 'C:\SampleProjects\py\mysite'. The Python interpreter section is expanded, showing 'New Virtualenv environment' selected. Under this, 'New environment using' is set to 'Virtualenv'. The location for the new environment is 'C:\Users\jetbrains\.virtualenvs\mysite'. The base interpreter is 'C:\Python36\python.exe'. There are checkboxes for 'Inherit global site-packages' and 'Make available to all projects', both of which are unchecked. Below this is the 'Previously configured interpreter' option, which is also unchecked. The interpreter is set to 'Python 3.8 C:\Users\jetbrains\AppData\Local\Programs\Python\Python38-32\python.exe'. The 'More Settings' section is expanded, showing 'Template language' set to 'Django', 'Templates folder' set to 'templates', and 'Application name' set to 'polls'. The 'Enable Django admin' checkbox is checked. A 'Create' button is at the bottom right.

New Project

- Pure Python
- Django**
- Flask
- Google App Engine
- Pyramid
- Web2Py
- Scientific
- Angular CLI
- AngularJS
- Bootstrap
- HTML5 Boilerplate
- React App
- React Native

Location: C:\SampleProjects\py\mysite

Python Interpreter: New Virtualenv environment

☒ New environment using Virtualenv

Location: C:\Users\jetbrains\.virtualenvs\mysite

Base interpreter: C:\Python36\python.exe

☐ Inherit global site-packages

☐ Make available to all projects

☐ Previously configured interpreter

Interpreter: Python 3.8 C:\Users\jetbrains\AppData\Local\Programs\Python\Python38-32\python.exe

More Settings

Template language: Django

Templates folder: templates

Application name: polls

☒ Enable Django admin

Create

Tassonomia di un'applicazione django

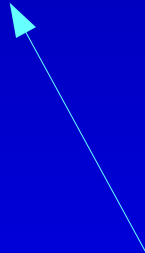
- **mysite/**: cartella top-level (contenitore di progetto)
- **manage.py**: script per automatizzare alcune operazioni di Django – wrapper attorno a django-admin.py
- **mysite/mysite/**: cartella del package che costituisce il corpo del progetto
- **__init__.py**: rende mysite/mysite un package
- **settings.py**: impostazioni per il funzionamento del progetto mysite, e definisce:
 - quali package importare
 - i dati sul DBMS da usare
 - ...

Tassonomia di un'applicazione django

- **wsgi.py**: file che serve da entry-point per l'interazione con un Web server (es. Apache, Nginx) esterno mediante standard WSGI
 - Interazione tra Web server e applicazioni Python
- **urls.py**: struttura dello spazio degli URL dell'applicazione - mappa gli URL richiesti in invocazioni verso le componenti del sistema (implementa il controller del progetto)
- File non presenti nello scheletro:
 - **models.py**: modello
 - **views.py**: view
- Sono spesso presenti solo nelle sotto-applicazioni (non di default) che svolgono azioni specifiche

Django-admin/manage

- Sintassi:
 - `manage.py <command> [options]`
- Principali comandi:
 - **startproject**: crea scheletro di un progetto
 - **startapp**: crea scheletro applicazione
 - **runserver**: avvia server Web di development
 - **migrate**: creazione e gestione DB
 - **dumpdata**: scarica dati DB
 - **loaddata**: carica dati in DB
 - **test**: esecuzione di test
 - ...



Proviamo che funzioni
con il server di
development

Proviamo il development server

- `python manage.py runserver`
- Possibile usare **porta diversa** dal default (8000) o **binding IP diverso** indicandolo mediante la linea di comando
 - Es. `python manage.py runserver 8080`

Output

Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, contenttypes, sessions.

Run 'python manage.py migrate' to apply them.

May 01, 2022 - 18:06:01

Django version 4.0.4, using settings 'mysite.settings'

Starting development server at `http://127.0.0.1:8000/`

Quit the server with **CONTROL-C**.

Tutto bene

A dopo per
il DB...

In ascolto
qui

Proviamo il development server

- Provare a collegarsi col browser a 127.0.0.1:8000
- 8000 porta di default del development server

django

[View release notes](#) for Django 2.1



The install worked successfully! Congratulations!

You are seeing this page because `DEBUG=True` is in your settings file and you have not configured any URLs.

Se avessimo
`DEBUG=False`

Not Found

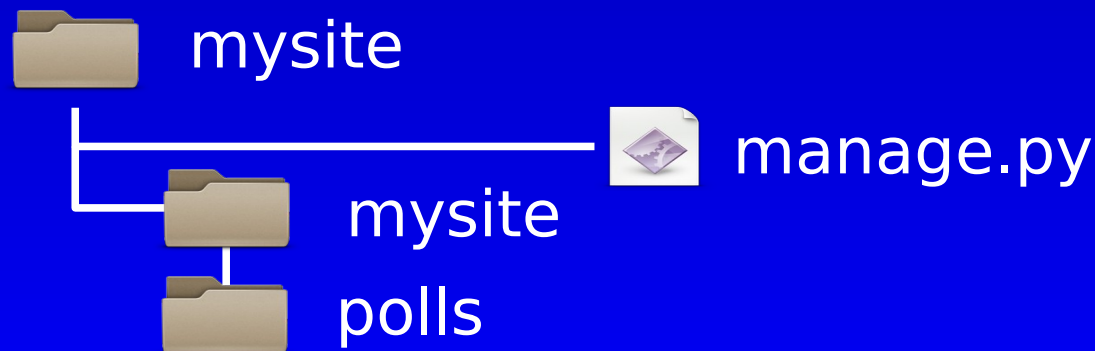
The requested URL / was not found on this server.

Creare una nuova applicazione

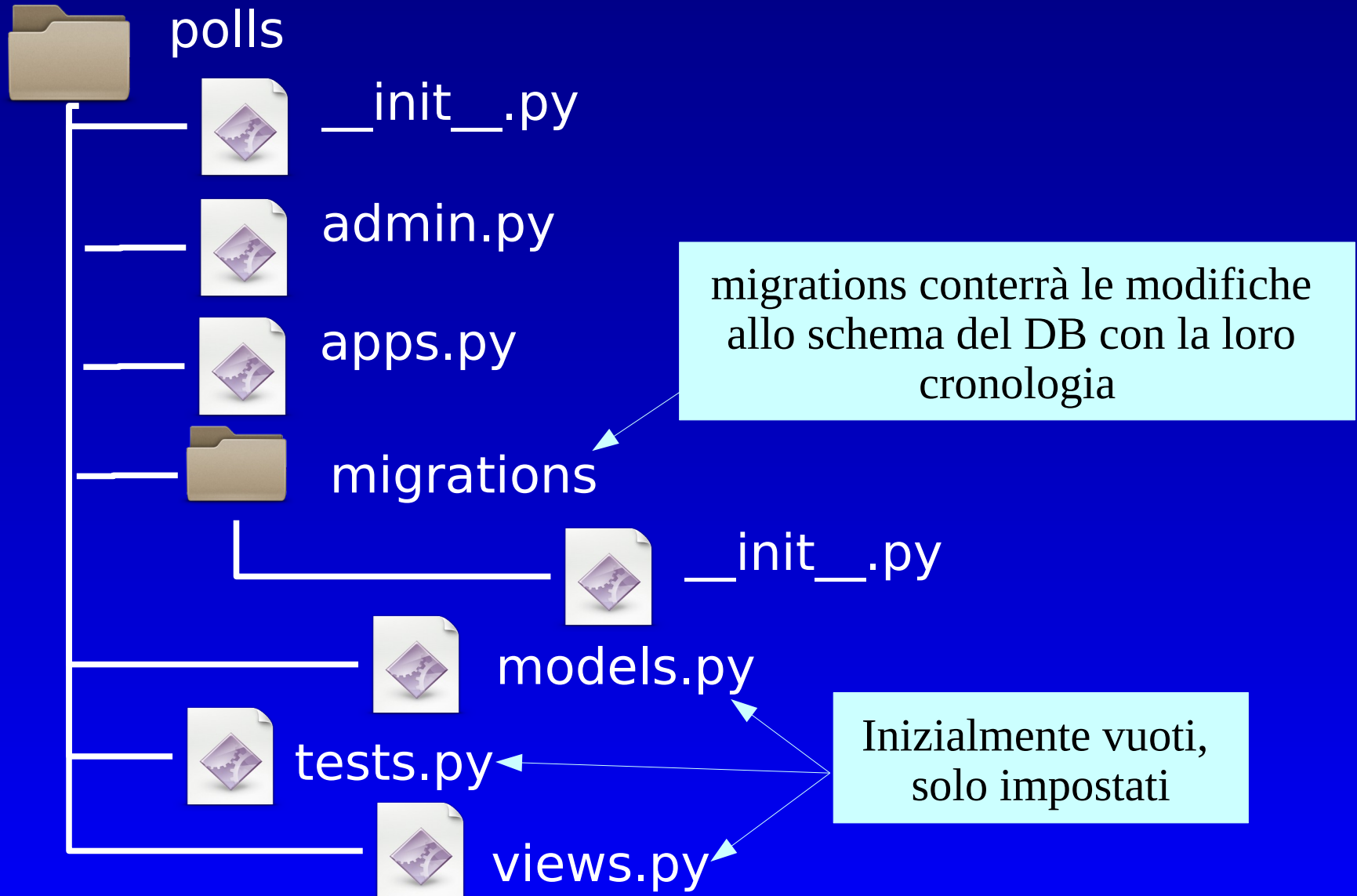
- Pratica comune: suddividere lo sviluppo del progetto in diverse applicazioni con funzionalità precise
- Es. per un sito di e-commerce:
 - Applicazione per gestione utenti
 - Applicazione per catalogo prodotto
 - Applicazione per acquisti online e fatturazione
- Applicazione = package Python con una specifica struttura di file e cartelle
 - Utility di Django per generare automaticamente la struttura della app
 - `python manage.py startapp nomeapp`

La nuova app polls

- Creare una **nuova app di nome polls**:
 - `$ python manage.py startapp polls`
- **Dove** lanciamo il comando (*dove mettiamo la directory polls*)?
 - Può essere **dovunque** nel Python Path
 - Noi la inseriamo allo stesso livello **di manage.py** per poterla importare come un modulo top-level

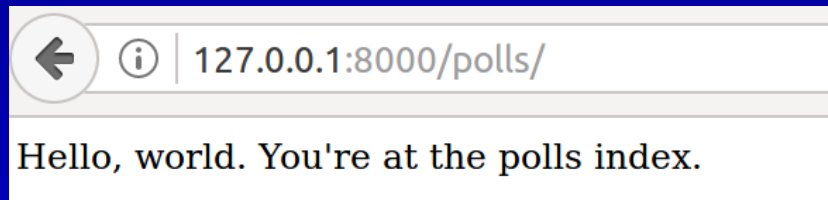


Struttura della nuova app



Una prima view

- All'indirizzo `http://127.0.0.1:8000/polls/` supponiamo di avere l'indice dei nostri sondaggi - per ora, vogliamo mostrare la scritta "Hello, world. You're at the polls index."



- Scriviamo la view: definita nel file `polls/views.py` come una **funzione Python**

```
from django.http import HttpResponse  
def index(request):  
    return HttpResponse("Hello, world. You're at the \\  
        polls index.")
```
- Ogni view deve **ritornare un oggetto di tipo HttpResponse** (*unico requisito indispensabile*)

Una prima view

- `def index(request):`

Parametro request:

- Oggetto di tipo **HttpRequest**: creato da Django ogni volta che viene richiesta una pagina (URL), poi passato alla view invocata
- Consente di accedere ai **dettagli (metadata)** della richiesta HTTP
- Scrivere la view non basta: per renderla **accessibile** dagli utenti attraverso il sito, bisogna **mapparla** ad un URL (controller)
 - Quale URL deve rispondere alla nostra view?
 - Abbiamo deciso di associare l'URL `127.0.0.1:8000/polls/`

Configurazione URLs

- **Controller** dell'applicazione: `mysite/urls.py`

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

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

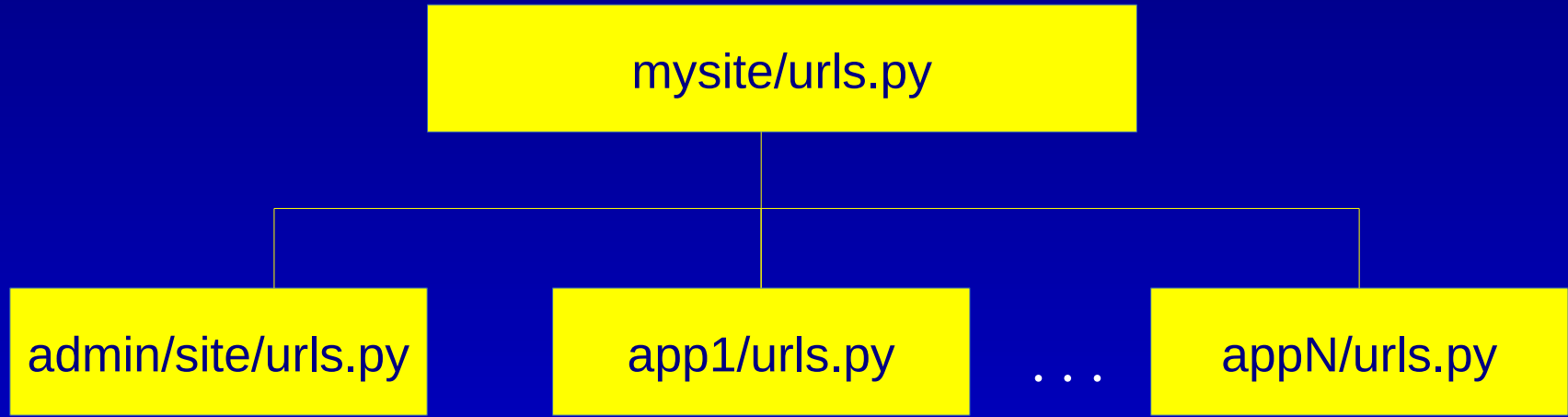
```
urlpatterns = [  
    path('admin/', admin.site.urls),  
]
```

- `urlpatterns = [...]` lista dei mapping tra URL → funzioni Python da chiamare (views)
- Unica riga presente → interfaccia di amministrazione `admin` (app/package importato)

File urls.py

- Quando arriva una richiesta all'indirizzo 127.0.0.1:8000/ Django cerca un **mapping nel file urls.py**
 - Specificato come ROOT_URLCONF in settings.py
`ROOT_URLCONF = 'mysite.urls'`
- Mapping tra pezzo di URL che segue host:port/
- Esempio già presente di “admin/”
`path('admin/', admin.site.urls),`
- `path()` riceve 2 argomenti obbligatori
 - **Primo parametro:** stringa che contiene un **URL pattern** che deve fare match con gli URL richiesti
 - **Secondo parametro:** indica come gestire le richieste che fanno match

Struttura gerarchica di urls.py



- Dopo aver trovato il mapping con la parte dell'URL che segue host:port/, si rimanda il controllo a quanto specificato nel controller della singola applicazione che compone il progetto
 - Es. admin/site/urls.py specifica le regole di matching per la parte di URL che segue 127.0.0.1:8000/admin

Rendere accessibile la view



- **Struttura gerarchica del controller**
- **Lo spazio di URL relativo a polls** deve essere gestito dal controller dell'applicazione → `polls/urls.py`
- Il controller dell'applicazione **polls** (`polls/urls.py`) deve essere incluso in `mysite/urls.py`
- Decidiamo che gli URL relativi a polls inizino in questo modo `http://127.0.0.1:8000/polls/`

Rendere accessibile la view

- #file mysite/urls.py

```
from django.urls import include, path
from django.contrib import admin
urlpatterns = [
    path('polls/', include('polls.urls')),
    path('admin/', admin.site.urls),
]
```

'polls/' fa match con
stringhe che iniziano
con polls/

'include' include altri file urls.py
Admin fa eccezione
(niente uso di include)

http://127.0.0.1:8000/polls/ → gestiti dal mapping in polls.urls

- Ora va specificato nel file polls/urls.py il mapping tra la funzione index() e l'URL 127.0.0.1:8000/polls/
 - mysite/polls/urls.py
 - File non presente, va aggiunto nella directory polls

Rendere accessibile la view

- Cosa dobbiamo mappare in polls/urls.py?
- Fino a “http://127.0.0.1:8000/polls/” è già considerato e rediretto: il controller di polls deve considerare solo la parte di URL eccedente questa

http://127.0.0.1:8000/polls/



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

```
urlpatterns = [
    path(' ', views.index, name='index'),
]
```

Niente spazio!

In polls/urls.py dobbiamo mappare la stringa vuota

Visitate

<http://127.0.0.1:8000/polls/>

Rendere accessibile la view

- `urlpatterns = [`
 `path(' ', views.index, name='index'),`
`]`



Analizziamo meglio
l'istruzione

- **4 parametri** passati a `path()`

- **Route (obbligatorio)**

stringa che contiene un URL pattern ' ': match di URL richiesti che iniziano con ' '

- **View (obbligatorio)**

- **Kwargs (opzionale)**

- **Name (opzionale)**

La struttura del comando url()

- **View:** funzione che viene chiamata quando si identifica un match con l'URL → `views.index`

Parametri passati alla view: request (HttpRequest) e ogni argomento (se presente) “catturato” dal parametro route come keyword arguments (vedremo...)

- **Kwargs:** parametro opzionale

Contiene lista di keyword arguments (chiave=valore) che possono essere passate alla view

- **Name:** parametro opzionale

Nome che serve per rendere **consistente e non ambiguo** il riferimento all'URL (disaccoppiamento del codice – spesso usato nei template)

Configurare il database

Ora andiamo a configurare il DBMS:
file `settings.py` – sezione **DATABASES**

```
DATABASES = {
```

```
    'default': {
```

```
        'ENGINE': 'django.db.backends.sqlite3',
```

```
        'NAME': BASE_DIR / 'db.sqlite3',
```

```
    }
```

```
}
```

ENGINE: tipo di DB

BASE_DIR = cartella
top level mysite

- **Default: SQLite**

- Non è un vero e proprio DB, ma tutti i dati vengono salvati su un file
- NAME è il path: `BASE_DIR/db.sqlite3`

Configurare il database

- Per usare **MySQL** o **PostgreSQL**, dovremmo creare un DB ed un utente che abbia su di esso tutti i diritti
- Quindi dovremo impostare nel file
 - Nome DB
 - Username
 - Password
- E l'ENGINE corrispondente (ad esempio)
'django.db.backends.postgresql',
'django.db.backends.mysql', or
'django.db.backends.oracle'

Sincronizzare il database

- Impostati i parametri per la connessione, è necessario **sincronizzare il database**:
 - `$ python manage.py migrate`
- L'operazione crea:
 - Il file **db.sqlite3** nella cartella corrente (sqlite3)
 - **Delle tabelle nel DB** (caso MySQL o Postgres)
- Verifica dati nel caso di SQLite3
 - `$ sqlite3` (potreste doverlo installare)
 - `sqlite> .open "db.sqlite3"`
 - `sqlite> .tables`

Sincronizzare il database

Per ogni applicazione elencata in settings.py sotto `INSTALLED_APPS`, il comando crea le relative tabelle

Tabelle relative alle app elencate in settings.py

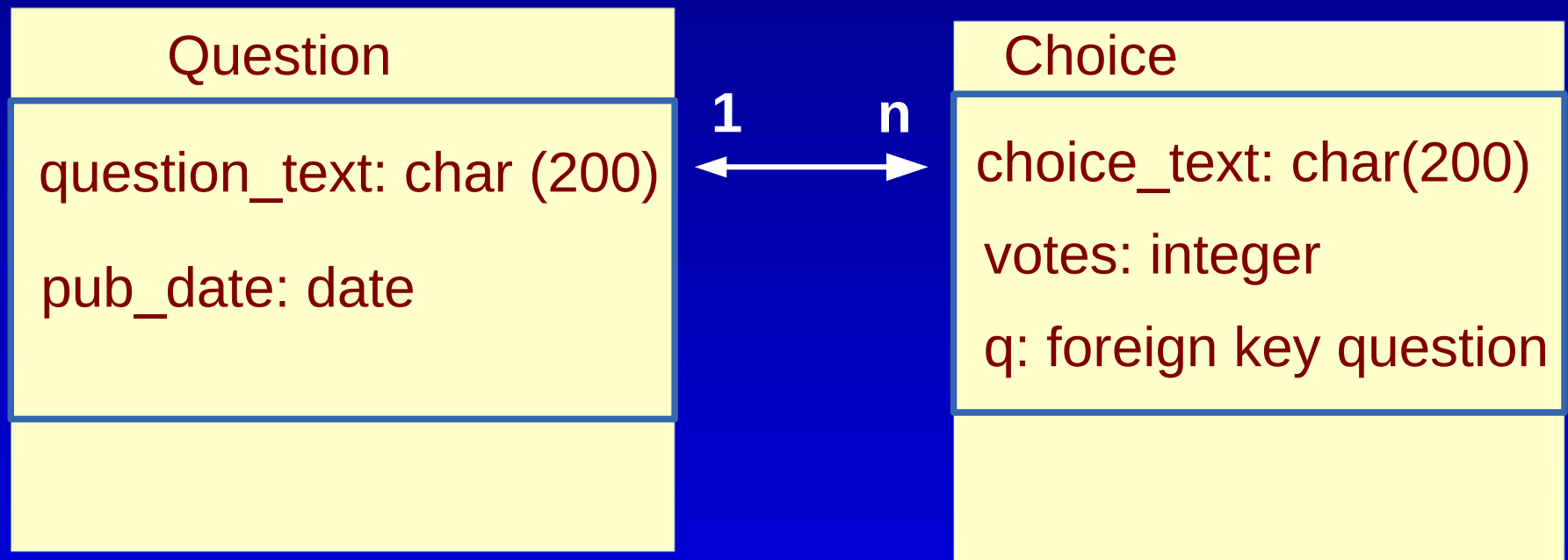
```
INSTALLED_APPS = (  
    'django.contrib.admin',    #sito di amministrazione  
    'django.contrib.auth',    #sistema di autenticazione  
    'django.contrib.contenttypes', #gestione content type  
    'django.contrib.sessions', #gestione sessioni utente  
    'django.contrib.messages', #gestione messaggi  
    'django.contrib.staticfiles', #gestione file statici  
)
```

Il modello dei dati

Modello della nuova app

- Il primo passo per scrivere una Web app che usa un database è **definire i modelli dei dati**
 - Schema del database con relativi metadati
- Sorgente unica dei dati - **principio DRY**
 - **Include le migrazioni** (modifiche allo schema del DB): derivate dai file del model, in una sorta di history che Django mantiene per mantenere la consistenza dei dati
- Nella app polls avremo due **modelli (classi Python)**:
 - **Question**: domanda e data di pubblicazione
 - **Choice**: testo della scelta e contatore di voti

Modello della nuova app



- Tabella = classe Python (sottoclasse di Model)
- Colonne (campi) della tabella = attributi di classe

File models.py

Question e Choice:
classi Python

Nomi degli attributi:
colonne tabelle DB

Tipi di dato dei campi
derivati da models

- File polls/models.py:

```
from django.db import models
```

```
# Create your models here.
```

```
class Question(models.Model):
```

```
    question_text = models.CharField(max_length=200)
```

```
    pub_date = models.DateTimeField('date published')
```

Argomento opzionale: nome human-readable

```
class Choice(models.Model):
```

```
    question = models.ForeignKey(Question,  
                                on_delete=models.CASCADE)
```

Rimozione di
campi in cascata

```
    choice_text = models.CharField(max_length=200)
```

```
    votes = models.IntegerField(default=0)
```

Obbligatorio

Opzionale

Analisi del modello

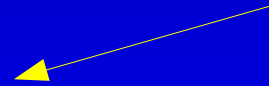
- **model.models** presenta una vasta lista di tipi di dato che possono essere rappresentati, ognuno con parametri specifici
- Le **relazioni** tra oggetto possono essere di vario tipo, come avviene comunemente nei DB:
 - Uno-a-uno
 - Uno-a-molti (es. Question → Choice)
 - Multi-a-molti
- Per le relazioni multi-a-molti: costrutti speciali attraverso attributi **Many-to-many field**
- Ogni tupla/record ha un **campo ID** usato come riferimento univoco (chiave surrogata numerica aggiunta in automatico)

Attivazione del modello

- Il codice in `models.py` dà informazioni per:
 - creare lo schema del DB per polls
 - creare le API Python per accedere agli oggetti Question e Choice
- Per attivare il modello, l'applicazione polls va aggiunta alle installed apps in `settings.py`
- `Settings.py`:

```
INSTALLED_APPS = (  
    'polls.apps.PollsConfig',  
    'django.contrib.admin',  
    [...]  
)
```

Aggiungiamo la classe di configurazione di polls



Sincronizzazione DB

`python manage.py makemigrations polls`

Migrations for 'polls':

0001_initial.py:

- Create model Question
- Create model Choice
- Add field question to choice

**ATT: non crea le tabelle
ma solo la migration!
File che contiene la
migration (numero 0001)**

- Makemigrations comunica che c'è stato un **cambiamento nel modello** che deve essere **memorizzato come una migration**
 - **Ogni migrazione** è un **file** human-readable (vedere `polls/migrations/0001_initial.py`)
 - Migrazione modellata come **classe Python**
`class Migration(migrations.Migration):`

Sincronizzazione DB

#polls/migrations/0001_initial.py

class Migration(migrations.Migration):

operations = [

migrations.CreateModel(

- name='Choice',

- fields=[

- ('id', models.AutoField(auto_created=True,

- primary_key=True, serialize=False,
verbose_name='ID')),

- ('choice_text',

- models.CharField(max_length=200)),

- ('votes', models.IntegerField(default=0)),

-], ...

Campi id di tipo
AutoField (primary_key)

Sincronizzazione DB

- Quale codice **SQL** viene generato da Django per apportare le modifiche al DB? Possiamo vederlo con:

```
$ python manage.py sqlmigrate polls 0001
```

- Nomi delle tabelle: polls_nomeclasse

- Es. polls_choice

- Chiave primaria: 'id' numerico

- '_id' aggiunto alla Foreign Key

- Per creare le tabelle

```
$ python manage.py migrate
```

- Migrate controlla se ci sono **migrazioni** che **non** sono ancora state **applicate** al modello (in tutte le app) e le esegue apportando i cambiamenti

Apply all migrations: admin, auth, contenttypes, polls, sessions

Applying polls.0001_initial... OK

**Comportamenti
modificabili**

Codice SQL generato

Nome tabella generata
sul DB

```
BEGIN;  
CREATE TABLE `polls_question` (  
  `id` AUTO_INCREMENT NOT NULL PRIMARY KEY,  
  `question_text` varchar(200) NOT NULL,  
  `pub_date` timestamp with time zone NOT NULL);  
CREATE TABLE `polls_choice` (  
  `id` AUTO_INCREMENT NOT NULL PRIMARY KEY  
  `choice_text` varchar(200) NOT NULL,  
  `votes` integer NOT NULL);  
ALTER TABLE `polls_choice` ADD CONSTRAINT  
  `poll_choice_question_id_246c99a640fbbd72_fk_polls_question_id`  
FOREIGN KEY (`question_id`)  
REFERENCES `polls_question` (`id`)  
DEFERRABLE INITIALLY DEFERRED;  
COMMIT;
```

ID aggiunti
automaticamente

Foreign key

**NOTA: il codice esatto
cambia a seconda del
DB in uso**

OSSERVAZIONI

- Le migrazioni rappresentano un **meccanismo di modifica del DB molto potente**
 - Possibilità di **modifica in corso d'opera senza perdita di dati**
 - Modifiche allo schema con dati inseriti!
 - Evita il bisogno di cancellare le tabelle e ricrearle per fare modifiche
- ***Riassunto passi da seguire per modifiche al model***
 - Cambiare il modello dei dati (models.py)
 - `python manage.py makemigrations [app]`
 - `python manage.py migrate`

Interagire con il modello attraverso le API Python

- **Per prove e verifiche:** possibile usare la **shell interattiva Python** fornita da `manage.py`
 - Variabili già opportunamente definite per lavorare con Django e il modello dati creato
- **\$ python manage.py shell**
- Setta i `DJANGO_SETTINGS_MODULE` e utilizza `mysite/settings.py`
- Dalla shell interattiva **possiamo importare i modelli appena creati** e usare alcune **API** per **accedere e modificare i dati nel DB**
 - API Python per accesso e modifica al DB
 - Proviamo ad usare qualche API che poi useremo nei file `views.py`

API Python

- Importiamo il modello

Import delle classi dal package polls

```
>>> from polls.models import Question, Choice
```

- Come accedere agli elementi nelle tabelle?

- Lista degli oggetti presenti

```
>>> Question.objects.all()  
[]
```

Per ogni classe la proprietà “objects” contiene la lista degli elementi della classe

- Creazione di una nuova question

```
>>> from django.utils import timezone
```

```
>>> q = Question(question_text="What's new?", \  
                  pub_date=timezone.now())
```

- Sincronizzazione esplicita con DB

```
>>> q.save()
```

Supporto per timezone abilitato di default (USE_TZ = True)

API Python

- **q** è un oggetto che rappresenta la mia tupla (entry)

```
>>> q.id
```

```
1
```

- **Accesso a campi dell'oggetto con notazione puntata**

```
>>> q.question_text
```

```
"What's new?"
```

- **Posso modificare i valori, chiamando però “save” alla fine**

```
>>> q.question = "What's up?"
```

```
>>> q.save()
```

- **Recupero degli oggetti question presenti**

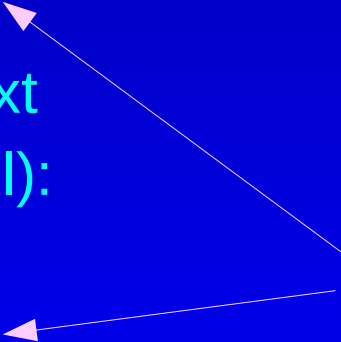
```
>>> Question.objects.all()
```

```
<QuerySet [<Question: Question object>]>
```

API Python

- Nota: non ho un meccanismo per **stampare in maniera utile** il riferimento a una tupla/record nel DB
- Ridefinire il **metodo speciale** che restituisce il riferimento
 - Metodo speciale `__str__()` : torna stringa UNICODE

```
class Question(models.Model):  
# ...  
def __str__(self):  
    return self.question_text  
class Choice(models.Model):  
# ...  
def __str__(self):  
    return self.choice_text
```



Sintassi dei normali
metodi di classe Python

Ulteriori metodi nel modello

Possiamo aggiungere altri **metodi** al modello che calcolino valori utili all'applicazione

- Es. Verifichiamo se una question è recente o no (pubblicata da meno di un giorno)

```
#File polls/models.py
```

```
import datetime
```

```
from django.utils import timezone
```

```
# ...
```

```
class Question(models.Model):
```

```
# ...
```

```
def was_published_recently(self):
```

```
    return self.pub_date >= timezone.now() - \
           datetime.timedelta(days=1)
```

Ritorna un booleano



Riprendiamo a giocare con la shell

- Lanciamo di nuovo la shell interattiva (necessario per poter ricaricare le modifiche) e verifichiamo

- Nota: non è necessaria nessuna migrazione

```
$ python manage.py shell
```

```
>>> from polls.models import Question, Choice
```

```
>>> Question.objects.all()
```

```
[<Question: What's up?>]
```

- Sfruttiamo le **API** per accedere/filtrare i dati senza bisogno di scrivere codice SQL
 - Accediamo attraverso **<Classe>.objects:** contiene la lista degli oggetti che possono essere acceduti attraverso apposite funzioni di **filtraggio**

Estrazione dati dal modello

- Possiamo interrogare la lista con `get()` per recuperare singoli oggetti (tuple)
 - Uso di chiavi primarie o campi unici
 - `q = Question.objects.get(id=1)`
 - `q.was_published_recently()`
- o creare delle query di filtraggio coi metodi `filter()` e `exclude()` per recuperare più di un oggetto
 - `Question.objects.filter(<predicato>)`: ritorna la *lista* degli oggetti filtrati sulla base del **predicato di filtraggio**
 - `Question.objects.exclude(<predicato>)`: ritorna la *lista* degli oggetti che non soddisfano il **predicato di filtraggio**

Predicati di filtraggio

- **Predicato di filtraggio:**
 - `<nomeCampo>__criterio=<valore>`
- **Esempio:**
 - `question_text__startswith='What'`
- **Possibili criteri:**
 - **Sottocampi** (es: `__year`, `__month` per le date)
 - **Substring** (es: `__startswith`, `__contains`)
 - **Operatori di confronto** (es: `__gte`, `__gt`)
 - **Uguaglianza** (`__exact`, oppure niente)

Doppio underscore



Esempi di estrazione dati

- **Esempi:**

Question.objects.get(id=10)
→ **Exception DoesNotExist**

```
>>> Question.objects.get(id=1)
```

```
[<Question: What's up?>]
```

```
>>> Question.objects.filter(question_text__startswith='What')
```

```
[<Question: What's up?>]
```

```
>>> current_year = timezone.now().year
```

```
>>> Question.objects.filter(pub_date__year=current_year)
```

```
<Question: What's up?>
```

- **Possibile riferirsi alla primary key - shortcut pk (equivale a id)**

```
>>> Question.objects.get(pk=1)
```

```
<Question: What's up?>
```

Shell interattiva

- Uso della classe choice: come selezionare le choice associate ad una question?

- **choice_set** = set creato automaticamente da Django per mantenere il collegamento 'inverso' rispetto ad una Foreign Key

- accedere a tutti gli elementi collegati

```
>>> q = Question.objects.get(pk=1)
```

```
>>> q.choice_set.all()
```

```
[]
```

**Relazione
one-to-many**

- **q.choice_set.create()** permette di creare delle choice associate a question

Creazione choice

- Creiamo alcune choice per la question q

```
>>> q.choice_set.create(choice_text='Not much', votes=0)
```

```
<Choice: Not much>
```

```
>>> q.choice_set.create(choice_text='The sky', votes=0)
```

```
<Choice: The sky>
```

```
>>> c = q.choice_set.create(choice_text='Just hacking  
                                again', votes=0)
```

```
<Choice: Just hacking again>
```

- La chiamata a `create()`: crea l'oggetto, esegue lo statement SQL INSERT, aggiunge l'oggetto al set di choice legate alla question q e lo ritorna

Shell interattiva

```
>>> q.choice_set.all()
[<Choice: Not much>, <Choice: The sky>,
 <Choice: Just hacking again>]
```



**Risultati sotto forma
di lista Python**

- Contiamo le choice associate a q

```
>>> q.choice_set.count()
3
```

- Nota: i dati sono già sincronizzati anche su DB – **create()** salva già direttamente sul database senza bisogno di chiamare **save()**

Shell interattiva

- Le API Python automaticamente seguono le relazioni senza bisogno di scrivere i JOIN espliciti
- ES: selezionare le choice relative a question che sono state pubblicate nell'anno corrente

Shell interattiva

- Le API Python automaticamente seguono le relazioni senza bisogno di scrivere i JOIN espliciti
- ES: selezionare le choice relative a question che sono state pubblicate nell'anno corrente

```
>>> Choice.objects.filter(question__pub_date__year =  
current_year)
```

```
[<Choice: Not much>, <Choice: The sky>, <Choice:  
Just hacking again>]
```

Nota: doppio underscore

- Non ci sono limiti: funziona su più livelli (più JOIN)

Rimozione di elementi

Posso rimuovere elementi con il metodo delete()

```
>>> c = q.choice_set.filter(choice_text__startswith =  
'Just hacking')
```

```
>>> c.delete()
```

Elemento rimosso direttamente dal DB

Fine prima parte....

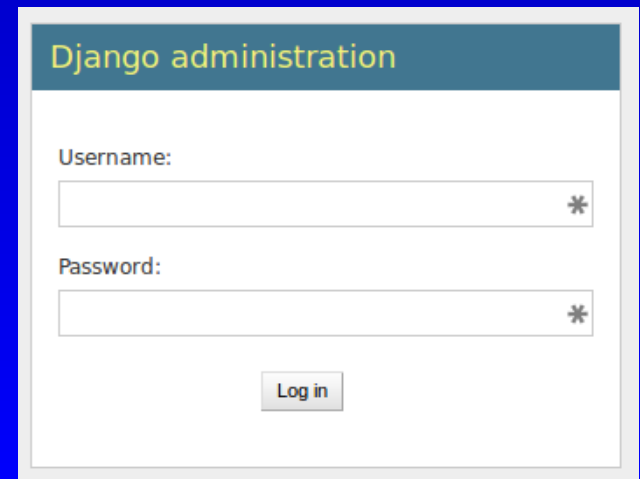
TUTORIAL DJANGO

Parte 2

Interfaccia Admin

Interfaccia di amministrazione

- Generata automaticamente da Django
- Inteso per essere utilizzato dai manager dei siti Web sviluppati e dai content publisher
- Permette di gestire tutti i modelli del progetto e i dati in essi contenuti
 - Utenti compresi e loro diritti
- L'interfaccia di amministrazione è attiva di default e visualizzabile a **127.0.0.1:8000/admin**

A screenshot of the Django administration login interface. It features a teal header bar with the text "Django administration". Below the header, there are two input fields: "Username:" and "Password:", each followed by a small asterisk icon. At the bottom of the form is a "Log in" button.

Django administration

Username: *

Password: *

Log in

Creare superuser

- Creare il superuser Django

```
python manage.py createsuperuser
```

Username: admin

Email address: admin@example.com

Password: ****

Password again: ****

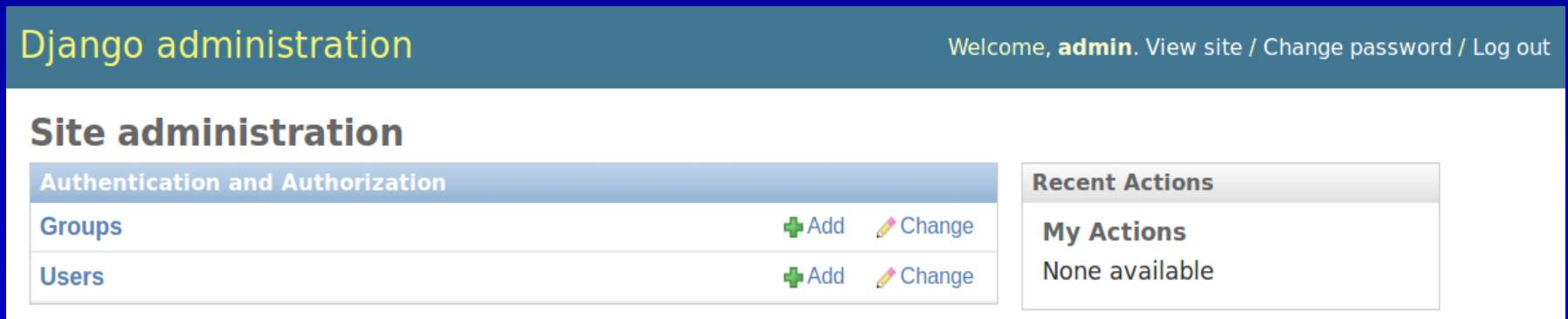
Superuser created successfully

Rilanciamo il server di development

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

Provare il lavoro fatto

- Dopo aver fatto il **login** si vede l'interfaccia di amministrazione con i campi Groups e Users forniti da `django.contrib.auth`



- Manca la nostra nuova applicazione polls
- **Abilitiamo la possibilità di modificarla** attraverso l'interfaccia di amministrazione

Abilitare interfaccia di amministrazione per polls

- Modificare il file `admin.py` (polls/admins.py)
`from django.contrib import admin`
`# Register your models here.`
- **Registriamo il modello Question** far fargli avere una interfaccia di amministrazione
`from .models import Question`
`admin.site.register(Question)`
- Ricarichiamo la pagina `127.0.0.1:8000/admin`
(non necessario riavviare il server se si modifica un file esistente)
- Esploriamo l'interfaccia di amministrazione

Django administration

Site administration

AUTHENTICATION AND AUTHORIZATION

Groups

[+ Add](#) [Change](#)

Users

[+ Add](#) [Change](#)

POLLS

Questions

[+ Add](#) [Change](#)

Recent actions

My actions

None available

Django administration

Home › Polls › Questions

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups

[+ Add](#)

Users

[+ Add](#)

POLLS

Questions

[+ Add](#)

Select question to change

0 questions

Views


Definizione di view in Django

- In Django **pagine Web e contenuti del siti** sono distribuiti attraverso il **meccanismo delle view**
- Una view tipicamente:
 - Viene invocata come conseguenza della richiesta di uno specifico URL
 - Click su un link, su un pulsante
 - Richieste HTTP GET e POST
 - Offre **funzioni specifiche** di logica applicativa (accede/filtra dati, effettua controlli ed elaborazioni)
 - Presenta un **proprio layout specifico (template)**
- Fase iniziale di progetto: definizione della **logica di navigazione**
→ set di views che risponderanno alle URL del nostro sistema

Nel nostro progetto polls

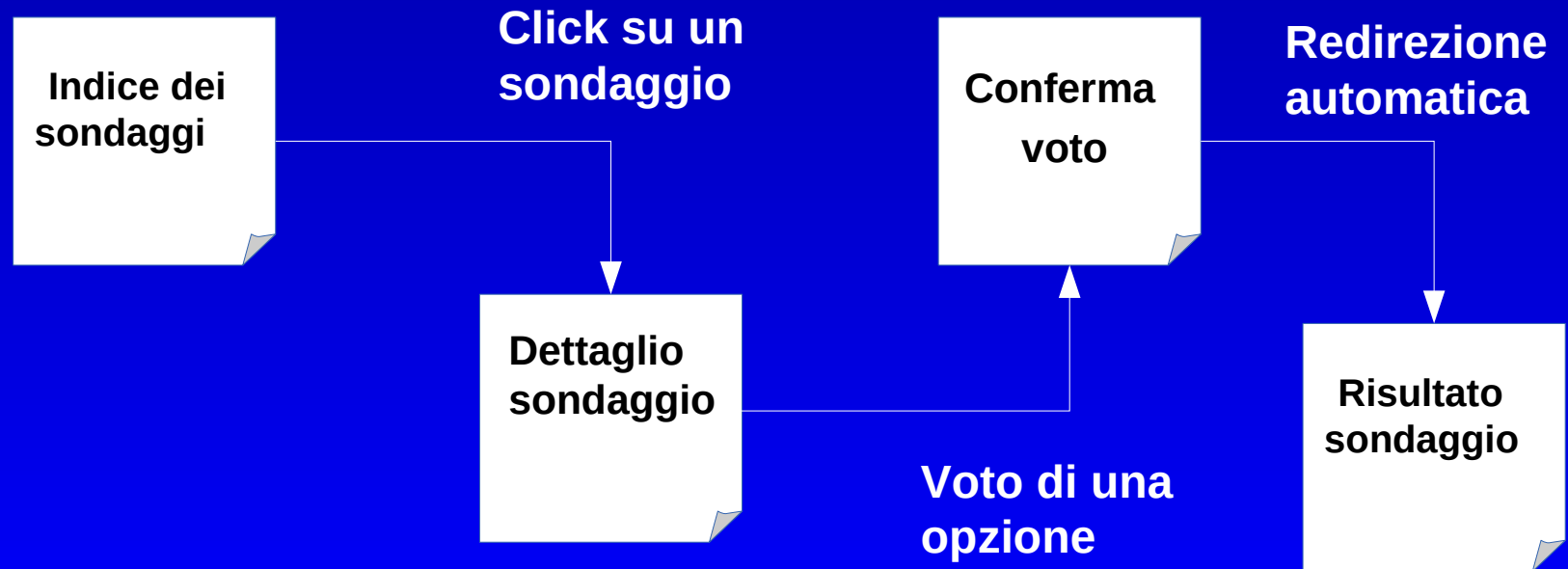
Definiamo 4 view iniziali per l'applicazione polls

- 1) **Indice dei sondaggi** → Indice degli ultimi 5 sondaggi caricati (dal più recente al più vecchio)
- 2) **Dettagli di un sondaggio** → pagina di dettaglio di un sondaggio, con le opzioni e la possibilità di votare (es. con un radio button)
- 3) **Pagina gestione/conferma voto** – gestisce e conferma l'operazione di voto per una choice specifica di un sondaggio
- 4) **Risultato di un sondaggio** → Pagina con i risultati di un sondaggio



Definire logica di navigazione

- **Indice dei sondaggi**
- **Dettagli di un sondaggio**
- **Pagina conferma voto**
- **Risultato di un sondaggio**



Views in Django

- Views: implementate attraverso **funzioni Python** nel file `views.py`
- Invoke dal controller in base all'**URL richiesto**
 - ES. `http://127.0.0.1:8000/polls/` → indice dei sondaggi
- Necessario **progettare gli altri URL** che risponderanno alle view
- Formato degli URL: Django **non usa le query string** per passare parametri
- Es.
`mysite/polls/polls.asp?year=2017&month=06&day=03`
- **Struttura** del tipo `mysits/polls/<year>/<month>/<day>/`



Definizione URL di polls

1)Indice dei sondaggi

`http://127.0.0.1:8000/polls/`

(view index impostata in `polls/views.py`)

2)Dettagli di un sondaggio

3)Pagina voto di un sondaggio

4)Risultato di un sondaggio

2) 3) e 4) si riferiscono
ad un sondaggio specifico

- Serve passare all'URL un parametro identificativo che permetta di recuperare il sondaggio dal DB
- Si usa la chiave primaria (id numerico)
 - Parametro passato attraverso l'URL

View di polls

Le view che corrispondono agli URL 2) 3) e 4) dovranno prendere in ingresso un parametro (question_id del sondaggio)

1) **Indice dei sondaggi** → **def index(request):**

http://127.0.0.1:8000/polls/

2) **Dettagli di un sondaggio** → **def detail(request, question_id):**

http://127.0.0.1:8000/polls/100/

3) **Pagina conferma voto** → **def vote(request, question_id):**

http://127.0.0.1:8000/polls/100/vote/

4) **Risultato di un sondaggio** → **def results(request, question_id):**

http://127.0.0.1:8000/polls/100/results/

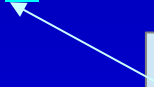
question_id di
uno specifico
sondaggio

Parametro passato
attraverso la struttura
dell'URL

Controller per gli URL di polls

- URL da collegare alla **view detail**
`http://localhost:8000/polls/100/`
- `http://localhost:8000/polls/` è già mappato in `mysite/urls.py`
- 100 rappresenta il **question_id** che deve essere “catturato” dal controller e **passato alla view**
- **Controller** per questo URL

```
path('<int:question_id>/', views.detail, name='detail')
```



Etichetta e tipo del parametro
recuperato dall'URL e
da passare alla view

es: `/polls/5/` dove 5 viene passato come `question_id`

NOTA: i parametri vengono passati automaticamente

Controller

file **urls.py** dell'applicazione polls (polls/urls.py)

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

```
from . import views
```

```
urlpatterns = [
```

```
    # es: /polls/
```

```
    path("", views.index, name='index'),
```

```
    # es: /polls/5/
```

```
    path('<int:question_id>/', views.detail, name='detail'),
```

```
    # es: /polls/5/results/
```

```
    path('<int:question_id>/results/', views.results, name='results'),
```

```
    # es: /polls/5/vote/
```

```
    path('<int:question_id>/vote/', views.vote, name='vote'),
```

```
]
```


View detail

- Definiamo le nuove view come altre **funzioni** nel file **polls/views.py**

```
def detail(request, question_id):
```

Dettagli su question_id

```
    return HttpResponse("You're looking at question %s." %  
                        question_id)
```

```
def results(request, question_id):
```

Risultati di question_id

```
    response = "You're looking at the results of question %s."  
    return HttpResponse(response % question_id)
```


```
def vote(request, question_id):
```

Voto su question_id

```
    return HttpResponse("You're voting on question %s." %  
                        question_id)
```

Validare il lavoro fatto

- Le view create si possono vedere rispettivamente agli URL di esempio:
- `http://127.0.0.1:8000/polls/`
- `http://127.0.0.1:8000/polls/100/`
- `http://127.0.0.1:8000/polls/10/results`
- `http://127.0.0.1:8000/polls/1100/vote/`



I parametri sono passati attraverso la struttura dell'URL
Non usano le query string

Inserire funzionalità nella view

- Ogni view è una funzione che deve avere obbligatoriamente uno di questi **due risultati**:
 - Restituire una **HttpResponse**
 - Sollevare una **eccezione** (es. errore tipo Http404)
- Per il resto, la view può fare tutto ciò che vogliamo usando quello che ci mette a disposizione il model
 - **Accesso ai dati** attraverso **le API Python** che abbiamo usato nella shell interattiva Django
- Funzionalità tipiche del codice di una view:
 - Interrogare il database ed elaborare dati
 - Usare/gestire i template da visualizzare
 - ...

Inserire funzionalità nella view

Riscriviamo la view index: che ritorni una lista degli ultimi 5 sondaggi inseriti in ordine cronologico inverso e separati da ', '

```
# File polls/views.py
from django.http import HttpResponse
from .models import Question
```

```
def index(request):
    latest_question_list = Question.objects.order_by('-pub_date')[:5]
    output = ', '.join([p.question_text for p in latest_question_list])
    return HttpResponse(output)
```

Accesso ai dati del DB:
ultimi 5 poll in ordine di
data decrescente



Stringa con gli ultimi 5 sondaggi separati da ', '



Verifichiamo
il funzionamento



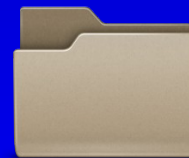
Osservazione

- Nota: il “design” della pagina è ora hardcoded nella vista
- Utilizziamo uno strumento più potente per separare il design della pagina dal codice della view: **template Django**
- Creiamo un template per la view index
- Obiettivo:

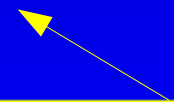
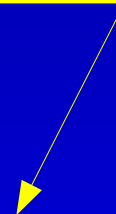
Visualizzare un elenco puntato di sondaggi (question_text) che siano dei link alle rispettive pagine di dettaglio (view detail)

Posizionamento dei template

Dove vanno inseriti i template di una app (polls) ?



Django cerca una directory
'templates' nella cartella
di ogni applicazione
(cartella da creare)



Dentro 'templates' creiamo
una cartella con il nome dell'app

Osservazione

File settings.py

```
TEMPLATES = [  
    {  
        'BACKEND': 'django.template.backends.django.DjangoTemplates',  
        'DIRS': [ ],  
        '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',  
            ],  
        },  
    ],  
]
```

In DIRS è possibile specificare altre cartelle in cui inserire i template

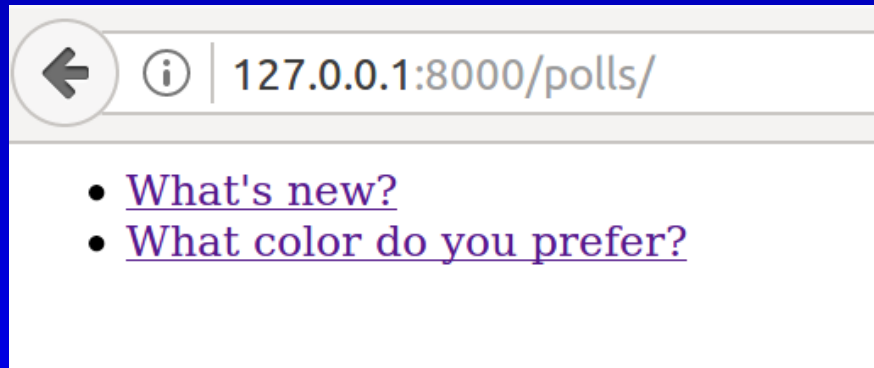
Posizionamento dei template

- Creazione di polls/templates/polls, dove inseriamo il file template index.html usato dalla view index
`polls/templates/polls/index.html`
- **Soluzione ottimale** per motivi legati a **modularizzazione e riuso** delle applicazioni
 - Dato il funzionamento del template loader, il file è riferibile nel codice come “polls/index.html”
- Perché un'altra cartella polls dentro templates?
Non basterebbe `polls/templates/index.html`?
- Sarebbe possibile, ma dovremmo riferirlo come “index.html”
- Se avessimo un altro template index.html in un'altra applicazione, ci sarebbe il rischio di confonderli (Django caricherebbe il primo che trova)

Visualizzazione desiderata

La pagina deve mostrare questa visualizzazione

- Elenco puntato dei sondaggi
- Ogni elemento dell'elenco mostri la Question_text del sondaggio e sia un link che porta alla pagina di dettaglio del sondaggio stesso
- 127.0.0.1:8000/polls/question_id/



- Se non ci sono sondaggi, scriva “No polls available”

Modificare le view

Modificare la view (polls/views.py) in modo che utilizzi il **template index.html** (che scriveremo)

```
from django.shortcuts import render
from .models import Question
```

Preimpostato nei
file views.py



```
def index(request):
    latest_question_list = \
        Question.objects.order_by('- pub_date')[:5]
    context = {'latest_question_list': latest_question_list}
    return render(request, 'polls/index.html', context)
```

Dictionary



Ritorna un
HttpResponse

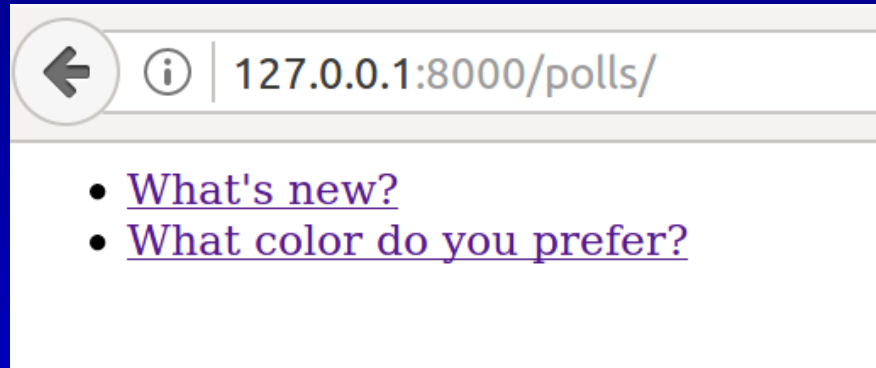


Componenti principali

- **Context:** dizionario per passare variabili Python al template
Mapping var_template → var_Python
Context = { var_template : var_Python }
- **Funzione render():** a partire da una data richiesta e da un context, ritorna un oggetto HttpResponse basato sul rendering del template passato in ingresso
- **render(request, template_name, context=None)**
- **Parametri e valore di ritorno di render()**
 - Request - richiesta HttpRequest
 - Template_name → nome del template
 - Context (opzionale) → dictionary con le variabili di cui il template deve avere visibilità
 - Ritorna un oggetto HttpResponse

Costruire il template

Template per visualizzare l'elenco dei sondaggi



- **Template Django:** mix di *HTML* e *comandi specifici detti tag*
- Esprimono la **presentazione** dei contenuti
- Caratteristiche: **potenza, semplicità e riuso**

Template di Django

Template: variabili

- `{{ title }}` rappresenta una **variabile** nel template Django – notazione `{{ NOMEVARIABILE }}`
- Variabili: usate per **scambiare dati/valori tra template e view di Django** attraverso il context
 - Context : dizionari con il mapping **variabile-valore**
- Quando in un template si incontra una **variabile**, ne viene valutato il **valore** che viene estratto dal **context**: la variabile viene sostituita con il valore corrispondente nel template

Template: variabili

Es.

- My first name is {{ first_name }}. My last name is {{ last_name }}.
- Context = {'first_name': 'John', 'last_name': 'Doe'}
- Risultato:
My first name is John. My last name is Doe.
- Se una **variabile non è definita**, Django non dà errore nel rendering del template ma la sostituisce di **default con stringa vuota ' '**

Template: filtri

- E' possibile manipolare l'output delle variabili mediante filtri
- I filtri sono definiti nei template attraverso l'uso di pipe “|”
Es. `{{variabile | filtro}}`
- Tra i filtri più usati:
 - `| default`: se la variabile è falsa o vuota viene mostrato il parametro del filtro (default)
`{{ value | default:"nothing" }}`
 - `| length`: mostra la lunghezza del dato (es. numero di elementi in una lista)
 - `| join`: `{{ list | join:", " }}` unisce con , e spazio

Template: tag e controllo di flusso

- I tag offrono la possibilità di inserire un po' di logica di programmazione nel processo di rendering e di produzione del contenuto
 - Identificati da {% ... %}
 - Possono accettare argomenti
- Possibilità:
 - produrre output
 - fornire strutture di controllo (if condizionali – cicli for)
 - estrarre contenuto da un database
 - abilitare accesso ad altri template tag

Template: controllo di flusso

- Tag con costrutti condizionali if:

```
{% if athlete_list %}
```

```
    Number of athletes: {{ athlete_list | length }}
```

```
{% else %}
```

```
    No athletes.
```

```
{% endif %}
```

```
---
```

```
{% if user.is_authenticated %}
```

```
    Hello, {{ user.username }}.
```

```
{% endif %}
```

- Iteratore for:

```
{% for x, y in points %}
```

```
    There is a point at {{ x }},{{ y }}
```

```
{% endfor %}
```

Possibilità di usare and e or
per creare condizioni complesse

Ipotesi: points è una lista di
tuple (x,y) passata al template
attraverso il context

Template: caratteristiche

- Caratteristiche di **ereditarietà e modularità**
- Un template “base” può essere descritto come un insieme di **blocchi** di codice del tipo
`{% block NOMBLOCCO %}{% endblock %}`
con funzione di placeholder
- Ogni template che eredita dal template base può ridefinire il blocco NOMBLOCCO, specializzandone il contenuto
- Quando un template **eredita** da un altro, i blocchi **ridefiniti** vanno a **sostituire** la definizione originale dei blocchi nel 'template padre'

Template: ereditarietà

- Esempio:

- Template base.html
- Template base_site.html che eredita da base.html

Definizione del blocco title (vuoto)

- **Base.html**

```
<title>{% block title %}{% endblock %}</title>
```

- **Base_site.html**

Tag: Eredita da base.html

```
{% extends "admin/base.html" %}
```

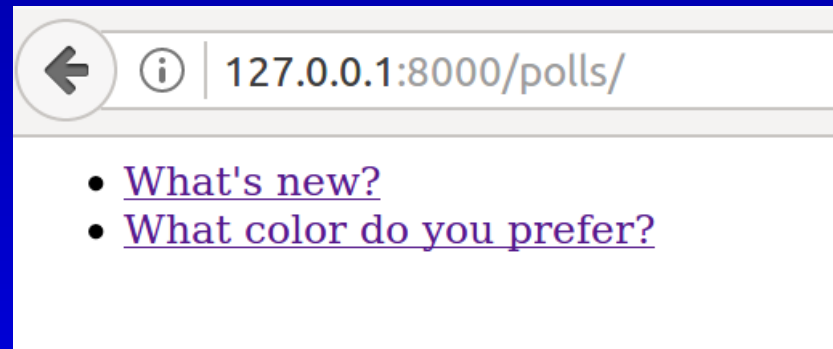
```
{% block title %}{{ title }} | {{ site_title |  
default:_('Django site admin') }}{% endblock %}
```

Ridefinizione del blocco title

Template di polls

Costruire il template

- Elenco puntato dei sondaggi
- Ogni elemento dell'elenco mostri la Question del sondaggio e sia un link che porta alla pagina di dettaglio del sondaggio stesso
 - `127.0.0.1:8000/polls/question_id/`



- Se non ci sono sondaggi, scriva “No polls available”
- Il template ha a disposizione la variabile `latest_question_list` → lista dei poll da visualizzare

Costruire il template

#File polls/templates/polls/index.html

```
{% if latest_question_list %}
```

```
<ul>
```

```
{% for question in latest_question_list %}
```

```
<li>
```

```
<a href="/polls/{{ question.id }}/">{{ question.question_text }}</a>
```

```
</li>
```

```
{% endfor %}
```

```
</ul>
```

```
{% else %}
```

```
<p>No polls are available.</p>
```

```
{% endif %}
```

In blu le parti HTML
In bianco il linguaggio dei
Template Django

Elenco puntato

Singolo punto elenco

Tag ancora

 Testo del link

Paragrafo

<p> </p>

Costruire il template

#File polls/templates/polls/index.html

```
{% if latest_question_list %}
```

```
<ul>
```

```
{% for question in latest_question_list %}
```

```
<li>
```

```
<a href="/polls/{{ question.id }}/">{{ question.question_text }}</a>
```

```
</li>
```

```
{% endfor %}
```

```
</ul>
```

```
{% else %}
```

```
<p>No polls are available.</p>
```

```
{% endif %}
```

Variabile `latest_question_list`
definita nella view `index` e
passata al template

Codice: `if` e ciclo `for`

Question id

Question_text

Link all'URL
`/polls/{{question.id}}/`
cui risponde la view
`detail` (dettaglio sondaggio)

Verifichiamo!

- Lanciamo il runserver
- Colleghiamoci all'URL
`http://127.0.0.1:8000/polls/`

View detail

- Possiamo ora ad occuparci di scrivere la **view detail**
 - **Risponde all'URL**
`http://127.0.0.1:8000/polls/question_id/`
 - **Pagina che mostra il dettaglio di un sondaggio**
 - Per ora **accontentiamoci** di mostrare il **question_text**
 - Possiamo al template l'oggetto **question**
- **Poniamoci una domanda: cosa accade se il sondaggio richiesto (question_id) non esiste?**
 - Il comportamento corretto è di inviare una risposta **HTTP che notifichi l'errore**
 - Es. HTTP 404 – Not Found

Gestione errori

La view detail (file polls/views.py) diventa:

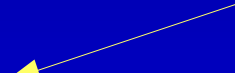
```
from django.shortcuts import get_object_or_404, render
```

```
from .models import Question
```

```
# ...
```

```
def detail(request, question_id):  
    question = get_object_or_404(Question, \br/>                                pk=question_id)  
    return render(request, 'polls/detail.html', \br/>                  {'question': question})
```

Exception sollevata
dal model se la
get() non trova nulla



Passo l'oggetto
question al template

Template associato: detail.html

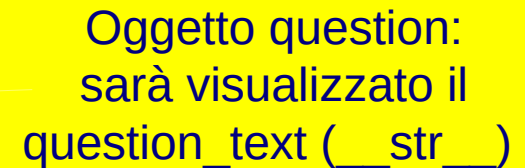
Percorso polls/templates/polls/detail.html

Per ora mettiamo soltanto

`<h1> {{ question }} </h1>`

Visualizzato come un titolo di livello 1

Oggetto question:
sarà visualizzato il
question_text (__str__)



E verifichiamo il comportamento: proviamo anche un
question_id non esistente

Es.

<http://127.0.0.1:8000/polls/4/>

Template detail.html

- Completiamo il template
- Vogliamo visualizzare oltre al question_text anche le choice associate – in un elenco puntato
- Il template viene passata nel context la variabile question (oggetto question estratto dal DB)
- Estraiamo le choice con question.choice_set

#polls/templates/polls/detail.html

```
<h1>{{ question.question_text }}</h1>
```

```
<ul>
```

```
{% for choice in question.choice_set.all %}
```

```
    <li>{{ choice.choice_text }}</li>
```

```
{% endfor %}
```

```
</ul>
```

Titolo h1

Elenco puntato

Chiamata a metodo:
interpretato come codice Python
question.choice_set.all() che
ritorna un oggetto iterabile (lista)

Rimuovere URL hard-coded dai template

- Possiamo rimuovere dal template index.html il riferimento hard-coded all'URL inserendo il riferimento a 'detail' (argomento name) in un tag {% url %}

```
# polls/templates/polls/index.html
```

```
...
```

```
{% for question in latest_question_list %}  
<li><a href="{% url 'detail' question.id%}">  
    {{ question.question_text }}</a></li>  
{% endfor %}
```

```
...
```

Tag {% url %}



Riferimento diretto all'URL detail attraverso name='detail':
il tag produce un URL formattato secondo le specifiche indicate in
polls/urls.py per la entry con name=detail, con parametro question.id
Accoppiamento lasco template-controller per facilitare modifica struttura URL

Namespace degli URL

- La struttura dei **nomi nei template** può causare **conflitti** se lo stesso nome di view è usato in app diverse dello stesso progetto
 - Es. view “detail” presente in altre app
- Come può Django differenziare **quale view detail chiamare** quando incontra il tag url `{% url 'detail' ... %}` ?
- Possibile introdurre dei **namespace per ogni app**

```
#file mysite/polls/urls.py
from django.urls import path
from . import views
app_name = 'polls'
urlpatterns = [
...

```

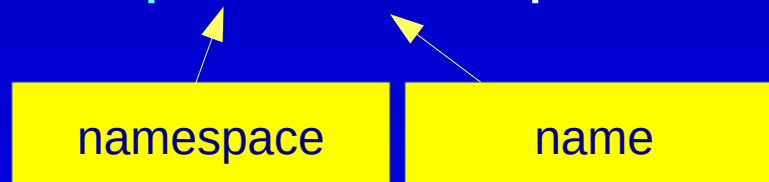
Namespace degli URL

- L'introduzione dei namespace deve essere riportata nei template relativi all'app, mettendo il namespace polls prima del nome della view

#file polls/templates/polls/index.html

 →

→



- In questo modo Django realizza la portabilità dei template

TUTORIAL DJANGO

Parte 3

Interazione con utenti

Gestire interazione con utenti

- Fornire agli utenti la **possibilità di esprimere un voto** su un determinato sondaggio
- Modifica al **template detail.html** per inserire questa possibilità – pagina di dettaglio del sondaggio
 - Visualizziamo le choice associate al sondaggio attraverso dei radio button
- Inseriamo un **pulsante** che permetta di **sottomettere la selezione della choice**
 - **Pulsante Vote**
- L'operazione di voto viene gestita **attraverso un form HTML**, inserito nel **template detail.html**



← 127.0.0.1:8000/polls/1/

What's now?

☐ Not much

☐ The sky

☐ Just hacking again

HTML Form e input type

- L'elemento `<form> </form>` è usato nelle pagine HTML per ricevere dati di input dall'utente
- Ogni elemento *form* contiene elementi `<input>` di tipo diverso a seconda del tipo di dato in input che dobbiamo ricevere
- Es. tipi di input
 - **text** Testo di input normale
 - **radio** Radio button (selezione di una tra più scelte)
 - **submit** Bottone per inviare i dati contenuti nella form
- *Esempio – text input type*

`<form>` Type e name parametri di `<input>`

name → Obbligatorio: serve per recuperare le informazioni

First name:
 `<input type="text" name="firstname">`

Last name:
 `<input type="text" name="lastname">`

`</form>`

`
` break line

First name:

Last name:

HTML e input type

Tutorial w3schools

<https://www.w3schools.com>

HTML input type

https://www.w3schools.com/html/html_form_input_types.asp

Describes the different input types for the <input> element, and many other things...

Provate gli input type text, submit, checkbox e radio

HTML Form e radio button

- *Esempio – radio + submit input type*

```
<form>
```

```
<label for="g1">Male</label>
```

value → valore associato al pulsante e passato come input se selezionato

```
<input type="radio" name="gender" id="g1" value="male"><br>
```

```
<label for="g2">Female</label>
```

```
<input type="radio" name="gender" id="g2" value="female">
```

```
<br>
```

```
<label for="g3">Other</label>
```

name → uguale per tutti i radio button per raggrupparli insieme

```
<input type="radio" name="gender" id="g3" value="other">
```

```
<br><br>
```

label for= → click sulla label per selezione

```
<input type="submit" value="Submit">
```

```
</form>
```

value → scritta sul bottone

Male ☐

Female ☐

Other ☐

Submit

Modifiche alla view dei dettagli

#polls/templates/polls/detail.html

```
<h1>{{ question.question_text }}</h1>
```

Elementi da definire nel template

- Form HTTP
- Un radio button per ogni choice associata al sondaggio
- Una label associata ad ogni choice (stesso id del radio button)
- Un pulsante Vote (submit)

Nota: ciclo for nel template

```
{% for choice in question.choice_set.all %}
```

Per associare un id diverso ad ogni choice
usiamo un contatore *forloop.counter*



The screenshot shows a web browser window with the address bar displaying '127.0.0.1:8000/polls/1/'. The main content area has the heading 'What's now?'. Below the heading are three radio button options: 'Not much', 'The sky', and 'Just hacking again'. At the bottom of the form is a 'Vote' button.

Modifiche alla view dei dettagli

#polls/templates/polls/detail.html

```
<form action="{% url 'polls:vote' question.id %}" method="post">
{% csrf_token %}
<fieldset>
  <legend><h1>{{ question.question_text }}</h1></legend>
  {% if error_message %}<p><strong>{{ error_message }}
                                </strong></p>{% endif %}
  {% for choice in question.choice_set.all %}
    <input type="radio" name="choice" id="choice{{ forloop.counter }}" \
      value="{{ choice.id }}">
    <label for="choice{{ forloop.counter }}">{{ choice.choice_text }}
    </label><br>
  {% endfor %}
</fieldset>
<input type="submit" value="Vote">
</form>
```

Form

Un radio button per ogni choice

Label per choice

Bottone submit 'Vote'

Spiegazione del codice: form

- Creo un form
- **Itero** sulle choice associate al sondaggio (question) di cui sto mostrando il dettaglio
- Per ogni choice presente creo:
 - **Un radio button**
 - **Una label** con il **testo** della choice (choice_text)
 - Radio button e label sono associati tramite un **id** basato sull'**iteratore del ciclo for** forloop.counter: contatore associato ai cicli del for
- Un **bottone** “Vote” di tipo submit, che finalizza il voto
- Il click sul pulsante Vote provoca la **action specificata nella form**

Spiegazione del codice: method

- Esaminiamo meglio la form: *method*

```
<form action="{% url 'polls:vote' question.id %}" method="post">
```

- **method="post"** : il click sul pulsante 'Vote' provoca l'invio di una richiesta HTTP di tipo **POST** (*default GET*)
 - i dati di input corrispondenti alla **selezione dell'utente** vengono inviati nel **body** della richiesta POST
- Come esattamente vengono inviati i dati nel body della richiesta POST?
 - Informazione sotto forma di coppia chiave=valore (dictionary)
 - Chiave è il valore specificato nel parametro name dell'elemento di input: **name="choice"**
 - Valore è il valore specificato nel parametro value dell'elemento di input: **value="{{ choice.id }}"** → identificativo id della choice selezionata

Spiegazione del codice: action

- Esaminiamo meglio la form - *action*

```
<form action="{% url 'polls:vote' question.id %}" method="post">
```

- La richiesta HTTP POST viene inviata ad un URL, che sarà associata ad una view per la gestione dei dati inviati
- Due alternative possibili
 - 1) Usare per la POST la stessa view (detail) che gestisce la richiesta GET modificandola opportunamente
 - 2) Usare una view differente appositamente creata
- Scegliamo di gestirli attraverso una view differente, la view `vote`, già predisposta nei file `polls/urls.py` e `polls/views.py`
- Ricordiamo che la view `vote` prende in ingresso il parametro `question.id`
 - `def vote(request, question_id):`

Spiegazione del codice: action

- Regola per fare match con l'URL associato a vote (polls/urls.py)
`path('<int:question_id>/vote/', views.vote, name='vote'),`
- Corrisponde a URL del tipo `polls/question_id/vote/`
- Uso di **accoppiamento lasco** attraverso il tag URL
`<form action="{% url 'polls:vote' question.id %}" method="post">`
 - URL non hardcoded: riferimento al “name” definito nel file `urls.py` e al namespace definito in `mysite/urls.py`
- Il **tag url** genera in output la **stringa** corrispondente all'URL (senza dominio) come specificato nel file `polls/urls.py` per la corrispondenza con `name='vote'`
 - Es. se `question_id=1` → genera: `1/vote/`
 - A `polls/1/vote` viene inviata la richiesta HTTP POST

Protezione contro CRSF

{% csrf_token %}

- Quando si utilizzano richieste POST, ci si deve difendere da attacchi **Cross-Site Request Forgeries (CSRF o XRSF)**
- **Schema di attacco CSRF:**
 - Si intercettano le attività di un utente che è già **autenticato** (cookies) sul sito
 - Si forza il browser dell'utente (es. con codice javascript) ad eseguire una **richiesta HTTP** di tipo POST non voluta (es. acquisto di un bene, trasferimento fondi, ...)
 - Il sito riceve una **richiesta HTTP** da un **utente trusted** e agisce di conseguenza
- Possibile tecnica per **iniettare dati malevoli**

Protezione contro CSRF

- Le richieste di tipo **POST** implicano una potenziale modifica dei dati → più pericolose
 - Importante **proteggersi** da questo tipo di attacco
- Il **tag** `{% csrf_token %}` inserito dentro le form produce un'informazione segreta che consente di identificare le **richieste 'fasulle'** che sfruttano la sessione utente
 - Input di tipo **hidden: non visibile sulla pagina**
`<input type='hidden' name='csrfmiddlewaretoken' value='E0nFyiPahN0Kd1TR0J2SwSy6Fs4tpDpF' />`
- Richieste POST che non provengono dalla stessa pagina (nel nostro caso: che non sono generate da interazione con la form nella pagina di dettaglio) sono **automaticamente rigettate**

HTML creato da view detail

- Sorgente della pagina <http://127.0.0.1:8000/polls/1/> (CTRL+U su Firefox)

<h1>What's now?</h1>

<form action="/polls/1/vote/" method="post">

question_id = 1

{% csrf_token %}

<input type='hidden' name='csrfmiddlewaretoken'
value='E0nFyiPahN0Kd1TR0J2SwSy6Fs4tpDpF' />

<input type="radio" name="choice" id="choice1" value="1" />

<label for="choice1">Not much</label>

<input type="radio" name="choice" id="choice2" value="2" />

<label for="choice2">The sky</label>

<input type="radio" name="choice" id="choice3" value="3" />

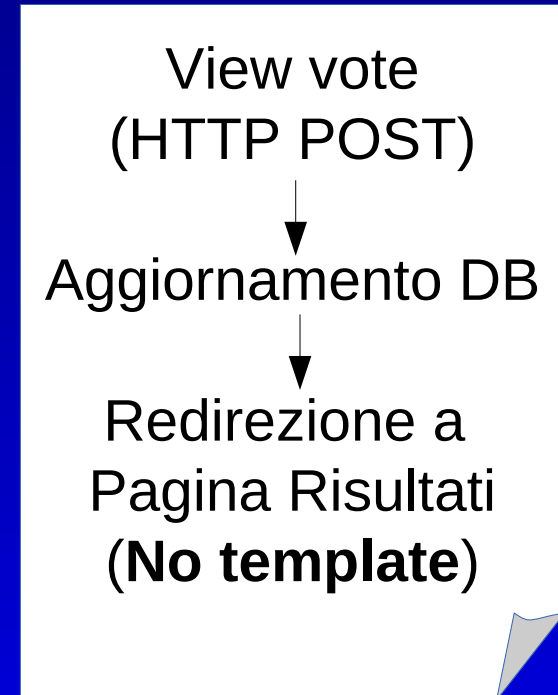
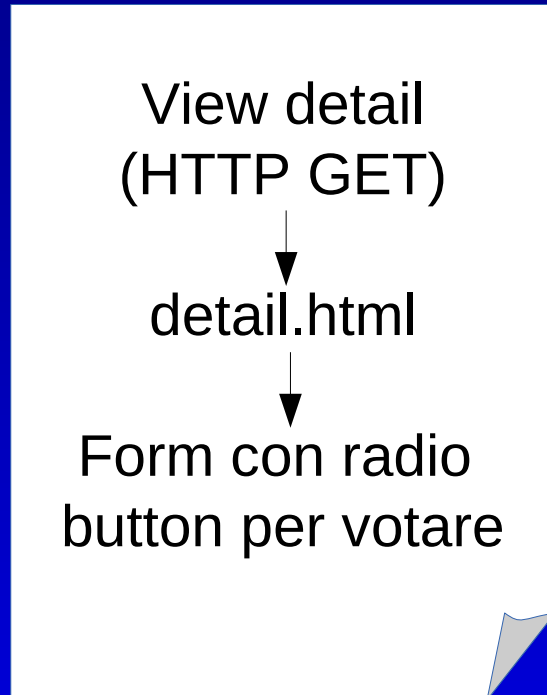
<label for="choice3">Just hacking again</label>

<input type="submit" value="Vote" />

</form>

Sostituzione choice_text

View vote: gestione del voto



Creazione della view vote

- View vote – URL polls/question_id/vote/
 - def vote(request, question_id):
- **Funzionalità richieste:**
 - **Estrazione** delle informazioni fornite in input: dati contenuti nel body della richiesta HTTP POST riguardo alla choice selezionata
 - **request.POST** è un **dictionary** che contiene i dati passati nel body della POST
 - **Notifica al modello** del risultato del voto: modifica dei dati nel DB – campo votes della choice selezionata
 - **Aggiornamento delle informazioni** mostrate all'utente: cambiamento della pagina visualizzata
 - Visualizzazione risultati del sondaggio (view results)

View vote – gestione errori

- **Errori possibili** che devono essere gestiti
 - L'utente **non ha effettuato nessuna selezione** prima di cliccare sul pulsante Vote
 - Dictionary request.POST non contiene l'elemento richiesto
 - La **choice selezionata non esiste** nel modello
 - Problema interno di passaggi dei valori (es. gestione errata del parametro value nei radio button del template)
 - polls/question_id/vote/ con question_id non presente
- **Reazione desiderata**
 - Mostrare di nuovo la form per il voto (template detail.html) in cui compaia però anche un messaggio di errore: “You didn't select a choice.”

Reazione desiderata

- Per ottenere la **reazione desiderata**
 - Mostrare di nuovo la form per il voto (template detail.html) in cui compaia però anche un messaggio di errore: “You didn't select a choice.”
- In caso di errore, devo richiamare il **rendering della pagina di dettaglio** – template detail.html
- Inserimento nel template detail.html di un controllo di errore:

```
{% if error_message %}  
  <p><strong>{{ error_message }}</strong></p> {% endif  
%}
```
- error_message: passato attraverso il context al template
- se error_message non è definito, l’if risulta falso

Creazione della view vote

```
from django.shortcuts import get_object_or_404, render
from django.http import HttpResponseRedirect, HttpResponse
from django.core.urlresolvers import reverse
from .models import Choice, Question
# ...
def vote(request, question_id):
```

question.id del sondaggio
su cui si è votato (passato
dal template detail.html)

```
    p = get_object_or_404(Question, pk=question_id)
    try:
```

```
        selected_choice = p.choice_set.get(pk=request.POST['choice'])
```

```
    except (KeyError, Choice.DoesNotExist):
```

Estraggo la choice
selezionata

```
        return render(request, 'polls/detail.html', {
            'question': p, 'error_message': "You didn't select a choice.",
        })
```

Se la choice non esiste, torno a
mostrare la form di voto (detail.html)

error_message
passato al template
detail.html

Creazione della view vote

```
# se try non solleva nessuna eccezione → in  
# selected_choice abbiamo l'oggetto choice estratto dal  
# DB che dobbiamo modificare
```

else:

```
selected_choice.votes += 1  
selected_choice.save()  
return HttpResponseRedirect(  
    reverse('polls:results', args=(p.id,))
```

Modifica dati nel DB

question_id

Redirezione alla view results: se il voto
è andato a buon fine, mostro all'utente i
risultati del sondaggio per cui ha appena votato

Spiegazione del codice: request.POST

- Struttura **request.POST**
 - `p.choice_set.get(pk=request.POST['choice'])`
 - Nel template: `<input type="radio" name="choice" value="{{ choice.id }}" />`
 - **Dictionary** che consente di accedere ai **dati sottomessi** attraverso la form (**name = value**)
 - `request.POST['choice']` → **choice.id** della scelta (**value**)
 - I valori estratti sono sempre di tipo **string**
- Implementazione di alcuni elementi di **controllo errori**
 - **KeyError** → generata se la chiave 'choice' non è presente nella richiesta POST (ES. caso di nessuna selezione!)
 - **Choice.DoesNotExist** → generato se selezioniamo una scelta non esistente nel modello (DB)

Spiegazione del codice

- **Gestione errori:**
 - In caso di errore (KeyError, Choice.DoesNotExist) torniamo a mostrare la pagina di dettaglio del sondaggio (template detail.html: vuole la variabile question) con un messaggio di errore contenuto nel campo **error_message**
 - `return render(request, 'polls/detail.html', { 'question': p, 'error_message': "You didn't select a choice."})`
 - Nel file detail.html:

```
{% if error_message %}  
    <p><strong>{{ error_message }}</strong></p> {% endif %}
```
- **Caso in cui non ci sono errori**
 - Si **incrementa il contatore** di voti nel modello
 - Si **salva** il valore
 - Si invoca una **redirezione a polls:results**

Spiegazione del codice

- Implementazione della **redirezione**

```
return HttpResponseRedirect( reverse('polls:results', args=(p.id,)))
```

- **HTTPResponseRedirect** al posto di **HTTPResponse**
- **HTTPResponseRedirect** vuole un solo **parametro**: l'**URL** che sarà oggetto della redirezione
- Funzione **reverse()** per trarre vantaggio dell'**accoppiamento lasco**: accetta il **nome della view** a cui si vuol passare il controllo e genera l'URL in base alle corrispondenti specifiche in **urls.py**
- Passiamo **esplicitamente** gli argomenti (ID del sondaggio) alla view usando il **parametro args** della funzione **reverse**
- Nel nostro esempio l'**URL generato** sarà **'/polls/question_id/results/'**

Best practice

- È pratica comune usare una **funzione di redirectione dopo aver processato dei dati di tipo POST** (buona pratica di programmazione Web)
 - PRG: Post Redirect GET per evitare doppi inserimenti (uso del bottone di 'back')
- La **funzione che gestisce la POST processa i dati**
 - Nel nostro caso la funzione vote
 - NOTA: vote() visualizza una pagina solo in caso di errore in cui torno a mostrare detail.html con messaggio di errore polls/question_id/vote/
- L'operazione di **visualizzazione delle informazioni dopo l'aggiornamento è delegata ad un altro URL**
 - Pagina dei risultati che visualizza il dato aggiornato
- Consente di migliorare la **modularità del codice**

Risultati di un poll

- L'operazione di voto termina con la **redirezione** alla pagina di risultati del sondaggio con i **dati aggiornati** (voti per ogni choice associata)
- La pagina dei risultati è generata dalla funzione **results** (che ora stampa solo una stringa) e dal corrispondente **template**, che chiameremo **results.html**
- Vogliamo generare una vista di questo tipo

“Vote again” deve riportare alla pagina di voto (detail)



Risultati di un poll

- Si tratta di scrivere codice piuttosto simile rispetto a quanto visto fino ad ora
- Il codice della **view results** sarà molto simile a quello della **view detail**
- Il codice del **template results.html** sarà simile – ma molto più semplificato - a quello del **template detail.html**

View Risultati

- Funzione results in polls/views.py:

```
def results(request, question_id):  
    question = get_object_or_404(Question,  
                                   pk=question_id)  
    return render(request, 'polls/results.html', \  
                  {'question': question})
```

Template risultati

- Struttura del template results.html

```
<h1>{{ question.question_text }}</h1>
```

```
<ul>
```

```
{% for choice in question.choice_set.all %}
```

```
    <li>{{ choice.choice_text }} -- {{ choice.votes }}
```

```
        vote{{ choice.votes|pluralize }}</li>
```

```
{% endfor %}
```

```
</ul>
```

Filtro pluralize: aggiunge un suffisso plurale (default 's') se choice.votes è maggiore di 1

```
<a href="{% url 'polls:detail' question.id %}">Vote again?</a>
```



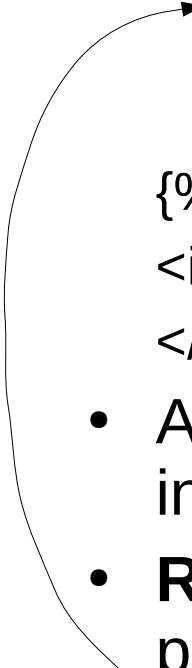
Django Forms



Form in Django

- Cosa è avvenuto per il voto su un sondaggio?
- **Creazione 'manuale' della form HTML** nel template detail.html

```
<form action="{% url 'polls:vote' question.id %}" method="post">
{% for choice in question.choice_set.all %}
    <input type="radio" name="choice" id="choice{% forloop.counter %}"
        value="{% choice.id %}" />
    <label for="choice{% forloop.counter %}">{% choice.choice_text %}</label>
{% endfor %}
<input type="submit" value="Vote" />
</form>
```



- Alla sottomissione della form (click su pulsante “Vote”) viene inviata una **richiesta POST alla view vote**
- **Recupero dei dati** attraverso il dictionary request.POST
pk=request.POST['choice']
- choice è il **name** dell'elemento di input radio button
(**value**=choice.id)

Uso di Form nelle pagine Web

- Tipico uso di form HTML: inserimento nuove 'entità'
 - Es. Registrazione nuovi utenti
- Creazione manuale della form: meccanismo **scomodo e complesso** per l'interazione con l'utente
 - Un elemento di input per ogni campo dell'oggetto da inserire, magari con widget dedicati
 - Per ogni elemento, ci sono controlli di validità (tipo/range) da effettuare
 - Meccanismo macchinoso di ricezione dei dati
 - Dictionary key-value ricevuto attraverso request.POST

Form in Django

- Possibilità di utilizzare gli oggetti **Form di Django**
- **Django Form:** oggetti composti da un insieme di campi che contengono informazioni su come **visualizzare** e **validare se stessi automaticamente**
 - **Sotto-classi di `django.forms.Form`**
- Un campo di una classe Form si mappa automaticamente su un elemento `<input>` della form HTML
- Ogni elemento della form **valida il dato relativo** quando la form è sottomessa
- Stesso meccanismo usato dall'interfaccia admin
- **Consigliato** inserirle in apposito file **`forms.py`** (ma non obbligatorio, possono stare anche in `models.py`)

Form in Django

- **Parecchi vantaggi:**
 - **Riutilizzo del codice** (non creo form dal nulla in ogni template)
 - Sfrutto le **capacità autodescrittive e autoanalizzanti** del codice (evito di scrivere parecchio codice ridondante e di controllo)
 - Posso **recuperare i dati dalle POST** in modo molto più agevole
- **Passi principali:**
 - Istanziare una form in una view
 - Form derivata da un modello (DB) o creata ad hoc
 - Form **Bound** o **Unbound** (con o senza dati già associati)
 - Passare la form al template attraverso il context
 - Espanderla nel template HTML: ad ogni elemento di input HTML corrisponde un campo della form)

Form in Django

- Consideriamo ad esempio di voler implementare una funzionalità '**Contattami**' sulla pagina del nostro sito
- All'URL 127.0.0.1:8000/polls/contact/ vogliamo inserire una form per recuperare i dati necessari
- Creiamo una view contact che mostra questa schermata
- Nel controller polls/urls.py
urlpatterns = [

 path('contact/', views.contact, name='contact'),
]

Subject:

Message:

Sender:

Cc myself: ☐

Form in Django

- Creiamo una **form Django** per facilitarci il compito

File **polls/forms.py**

```
from django import forms
```

```
class ContactForm(forms.Form):
```


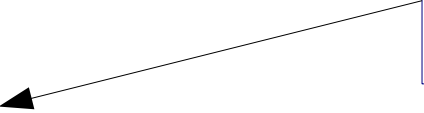
```
    subject = forms.CharField(max_length=100)
```

```
    message = forms.CharField()
```

```
    sender = forms.EmailField()
```

```
    cc_myself = forms.BooleanField(required=False)
```

Linguaggio dichiarativo
simile a quello usato
nei modelli Django
Specifica dei tipi di campo



Campi che compongono
la ContactForm

Comportamento voluto

- Come vogliamo che avvenga la gestione della richiesta POST che viene inviata al click sul pulsante SUBMIT?
 - Ricordiamo: può essere gestita 1) dalla stessa view che gestisce la GET oppure 2) si può creare un'ulteriore view che la gestisca
 - Scegliamo la prima opzione
- Quindi la view contact deve gestire:
 - Richiesta HTTP GET: visualizzazione della form
 - Richiesta HTTP POST: processamento dei dati sottomessi e gestione comportamento successivo

Usare la form nella view

#File polls/views.py

Vediamo come usare la form
nella **view contact()**
associata a /polls/contact/

...

from polls.forms import *

Import della form

def contact(request): #definizione della view contact

if request.method == 'GET': # Richiesta GET --> visualizziamo la form

form = ContactForm() # An unbound form

return render(request, 'polls/contact.html', {'form': form, })

elif request.method == 'POST':

Richiesta POST --> vedremo dopo

Passa la form unbound al
template contact.html
che andremo a scrivere

Istanza vuota
Unbound form: non ha dati
associati da DB – campi
vuoti o con i default

Template contact.html

- Scrivere il **template** è molto **semplice**

<form method="post">

No action:POST allo stesso URL
a cui risponde la view contact

- {% csrf_token %}

{{ **form.as_p** }}

Utilizzo della form

<input type="submit" value="Submit" />

</form>

Template contact.html

- Scrivere il **template** è molto **semplice**

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

No action: POST allo stesso URL
a cui risponde la view contact

```
{{ form.as_p }}
```

```
<input type="submit" value="Submit" />
```

Utilizzo della form

```
</form>
```

- **form.as_p** renderà automaticamente la form coi suoi campi come:

```
<p><label for="id_subject">Subject:</label>
```

Name uguali a field_name

```
<input type="text" id="id_subject" name="subject" maxlength="100" /></p>
```

```
<p><label for="id_message">Message:</label>
```

Vincolo di default

```
<input type="text" name="message" id="id_message" /></p>
```

Label ricavati
da field_name

```
<p><label for="id_sender">Sender:</label>
```

```
<input type="email" name="sender" id="id_sender" /></p>
```

ID posto a
id_field-name

```
<p><label for="id_cc_myself">Cc myself:</label>
```

```
<input type="checkbox" name="cc_myself" id="id_cc_myself" /></p>
```


Customizzare e processare dati

- Se vogliamo **customizzare** il template, ogni singolo campo della form è accessibile nel template come variabile
`{{ form.field-name }}`

- Vediamo ora come processare i dati recuperati dalla richiesta **POST** nella view **contact()**

```
def contact(request): #view contact
```

```
...
```

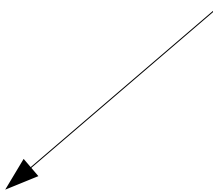
```
elif request.method == 'POST':
```

```
form = ContactForm(request.POST)
```

```
#form bound ai dati contenuti nella richiesta POST
```

```
if form.is_valid(): #Step di validazione dei dati
```

Istanza bound (con dati)
Recupero i dati già divisi
nei campi appositi



Validare i dati

- if **form.is_valid()**
 # **Process** the data in form.cleaned_data
 ...
• La funzione **form.is_valid()** esegue il controllo sulla validità dei dati contenuti nella form bound (inseriti dall'utente)
 - Controllo automatico sui diversi campi
 - Ritorna True in caso di successo
- Dopo aver chiamato (con successo) **form.is_valid()**, i **dati validati** saranno direttamente accessibili attraverso il **dictionary form.cleaned_data**
 - La key di accesso è il nome del campo della form
 - I dati sono stati convertiti in formati accessibili da Python

Processare i dati

```
if form.is_valid():
```

```
    subject = form.cleaned_data['subject']  
    message = form.cleaned_data['message']  
    sender = form.cleaned_data['sender']  
    cc_myself = form.cleaned_data['cc_myself']
```

Dictionary
form.cleaned_data



```
    recipients = ['info@example.com']
```

Destinatari della email



```
    if cc_myself:
```

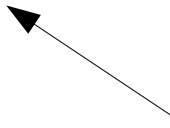
```
        recipients.append(sender)
```

Aggiungo il sender tra i
destinatari



```
return HttpResponseRedirect('/thanks/') # Redirect after POST
```

Redirezione a ipotetica
pagina di conferma



View contact

#File portal/views.py

```
from django.shortcuts import render
from django.http import HttpResponseRedirect
from portal.forms import *
```

```
def contact(request): #definizione della view contact
    if request.method == 'GET': # solo visualizzazione form
        form = ContactForm() # An unbound form
        return render(request, 'portal/contact.html', {
            'form': form,
        })
    elif request.method == 'POST': # se la form è stata sottomessa
        form = ContactForm(request.POST) #form bound ai dati della POST
        if form.is_valid(): # validazione passata
            # Processamento dei dati in form.cleaned_data
            return HttpResponseRedirect('/thanks/') # Redirect a /thanks/
```

Unbound form: non ha dati associati da DB – campi vuoti o con i default

Verifichiamo il comportamento e la validazione dei dati inseriti

Creare form dai modelli

- Nelle applicazioni Web basate su database, può capitare spesso di avere **form** per inserire/modificare entità del DB: **i campi form mappano quelli dei relativi modelli** del DB
- Es. Modello **BlogComment**: creare una form che permette all'utente di caricare un nuovo commento al blog
 - Ridefinire tutti i campi e i relativi tipi di dato nella form sarebbe **ridondante (già definiti nel model)**
- **ModelForm class: crea una form direttamente da un Django Model** con tutti i campi che è possibile inserire o modificare (non comprende i campi auto-generati – es. id)
- Nel caso di **Foreign Key** i campi generati saranno del tipo **MultipleChoiceField**
 - Presenterà una **scelta obbligata** tra elementi esistenti

Creare form dai modelli

```
>>> from django.forms import ModelForm
>>> from myapp.models import Article
```

Supponiamo di avere un **model Article** con i campi `pub_date`, `headline`, `content`, `reporter`

```
# Create the form class / file forms.py
```

```
>>> class ArticleForm(ModelForm):
...     class Meta:
...         model = Article
...         fields = ['pub_date', 'headline', 'content', 'reporter']
```

La classe **ArticleForm** generata avrà un campo per ogni campo del model **Article** specificato nell'ordine nella lista **fields**

```
#file views.py - FASE GET
```

```
# Creating a form to add an article
```

```
>>> form = ArticleForm()
```

```
# Creating a form to change an existing article.
```

```
>>> article = Article.objects.get(pk=1)
```

```
>>> form = ArticleForm(instance=article)
```

Creazione istanza di ArticleForm:
unbound form

Creazione di bound form con i
dati di un articolo esistente (pk=1)

Esempio completo – App Books

Esempio di applicazione books con 2 modelli

1) Author : modella entità autore

Campi : name, title (Mr., Mrs., Ms.), nationality (opzionale)

2) Book: modella entità libro

Campi: name, authors

- Nota: il campo title di Author è vincolato alle opzioni indicate:
Parametro **choices** specificato nel campo title
- Relazione: **molti-a-molti** tra Author e Book
Attributo **ManyToManyField** per modellare relazione molti-a-molti

```
$ python manage.py startapp books
```

```
#file mysite/mysite/urls.py:
```

```
path('books/', include('books.urls', namespace="books"))
```

Many-to-Many fields



Relazione multi-a-molti

Una relazione multi-a-molti di tipo semplice (senza attributi) non si modella con una relazione (tabella separata) ma sfruttando un **campo speciale dei modelli Django**

- Campo **ManyToManyField** in uno dei modelli coinvolti
 - Richiede come parametro il nome dell'altro modello coinvolto nella relazione multi-a-molti

Esempio completo - Modelli

```
#file books/models.py  
from django.db import models
```

```
TITLE_CHOICES = ( ('MR', 'Mr.'), ('MRS', 'Mrs.'), ('MS', 'Ms.') )
```

Tupla di tuple

```
class Author(models.Model):  
    name = models.CharField(max_length=100)  
    title = models.CharField(max_length=3, choices=TITLE_CHOICES)  
    nationality = models.CharField(max_length=100, blank=True)  
    def __str__(self):  
        return self.name
```

Accetta un
Oggetto iterabile

Serve per la visualizzazione

```
class Book(models.Model):  
    name = models.CharField(max_length=100)  
    authors = models.ManyToManyField(Author)
```

Identifica una **relazione multi-a-molti con il model specificato**. Equivale a multiple ForeignKey

Forms - books/forms.py

```
from django.forms import ModelForm ; from .models import Author, Book
```

```
class AuthorForm(ModelForm):
```

```
    class Meta:
```

```
        model = Author
```

```
        fields = ['name', 'title', 'nationality']
```

```
class BookForm(ModelForm):
```

```
    class Meta:
```

```
        model = Book
```

```
        fields = ['name', 'authors']
```

Definizione delle form con i nomi dei campi dei rispettivi modelli. Ogni campo del model ha **automaticamente associato un tipo di campo** della form

**Alternativa scritta
“a mano”**

```
class AuthorForm(forms.Form):
```

```
    name = forms.CharField(max_length=100)
```

```
    title = forms.CharField(max_length=3,  
                           widget=forms.Select(choices=TITLE_CHOICES))
```

```
    nationality = forms.CharField(max_length=100, required=False)
```

Widget = box di
selezione delle scelte
presenti in choices

```
class BookForm(forms.Form):
```

```
    name = forms.CharField(max_length=100)
```

```
    authors = forms.ModelMultipleChoiceField(queryset=Author.objects.all())
```

Multiple Foreign Key →
Scelte obbligate tra tutte le
entità presenti in Author

Lista di conversioni tra campi

Conversioni automatiche tra campi del model e campi della form

Model field

AutoField
CharField
DateField
DateTimeField
FileField
ForeignKey
ImageField
IntegerField
IPAddressField
ManyToManyField
NullBooleanField
TextField
URLField

Form field

Not represented in the form
CharField with max_length = model field's max_length
DateField
DateTimeField
FileField
ModelChoiceField
ImageField
IntegerField
IPAddressField
ModelMultipleChoiceField
NullBooleanField
CharField with widget=forms.Textarea
URLField

ID numerico

Casi speciali

Uso della form nelle view

2 casi tipici:

- 1) Creazione nuovo elemento → Uso di unbound form nella view
- 2) Modifica elemento esistente → Uso di bound form nella view

1) Creazione di un nuovo elemento

#file books/views.py

```
from .models import Author, AuthorForm
```

```
def insertAuth(request):
```

```
    if request.method == 'GET': # GET request: just visualize the form
```

```
        form = AuthorForm() # An unbound form
```

```
        return render(request, 'books/insertForm.html', { 'form': form,})
```

```
    else: # POST request
```

```
        pass
```

Uso della form nelle view

2) Modifica elemento esistente

#file books/views.py

```
from .models import Author, AuthorForm
```

Id autore da modificare

```
def modifyAuth(request, a_id):  
    if request.method == 'GET': # GET request: just visualize the form  
        a = Author.objects.get(pk=a_id)  
        form = AuthorForm(instance=a) # A form bound to instance a  
        return render(request, 'books/insertForm.html', { 'form': form, })  
    else: # POST request  
        pass
```

Il template resta lo stesso

Controller e template

#file books/urls.py

```
from django.conf.urls import url
from . import views
urlpatterns = [
    # ex: /books/insertAuth/
    path('insertAuth/', views.insertAuth, name='insertAuth'),
    # ex: /books/modifyAuth/1/
    path('modifyAuth/<int: a_id>/', views.modifyAuth, name='modifyAuth'),
]
```

Id autore a_id

#file /books/templates/books/insertForm.html

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

La POST viene mandata
allo stesso URL

Visualizzazione automatica
della form

Settings e migration

```
#mysite/setting.py
INSTALLED_APPS = [
    'polls.apps.PollsConfig',
    'books.apps.BooksConfig',
    ...
```

Verifichiamo


<http://127.0.0.1:8000/books/insertAuth/>

Modifica al DB – devo lanciare migrate

\$ python manage.py makemigrations books

\$ python manage.py migrate

Tendina per scelta tra opzioni



Name:

Title:

Nationality:

Submit

Recupero dati: metodo `form.save()`

- Ogni `ModelForm` ha un metodo `save()` che **crea e/o modifica** un oggetto sul database partendo dai dati disponibili su una **bound form** -> **recupero veloce dei dati da una POST**

```
from django.http import HttpResponseRedirect
```

```
from .models import Author
```

```
from .forms import AuthorForm
```

```
def insertAuth(request):
```

```
...
```

```
else: #POST request
```

```
    #Create a form instance from POST data.
```

```
    f = AuthorForm(request.POST)
```

```
    # Save a new Author object from the form's data.
```

```
    new_author = f.save()
```

```
    return HttpResponseRedirect('New Author Saved')
```

View che risponde a
POST di inserimento Author

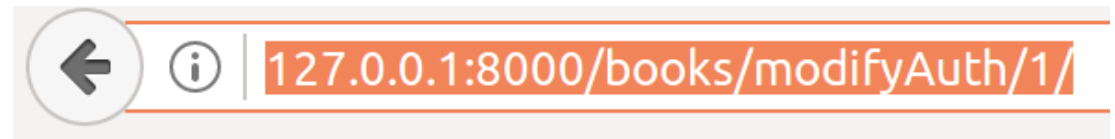
Recupero i dati inseriti
dall'utente dalla richiesta
POST attraverso la form:
form bound sui dati
inseriti dall'utente

Creazione di un
nuovo Author sul DB

Inserimento e Verifica

- Ora è possibile inserire un nuovo autore
- E successivamente verificare la view di modifica dell'autore appena inserito

Verifichiamo
<http://127.0.0.1:8000/books/modifyAuth/1/>



Mostra la form bound
all'istanza di autore appena
Inserita (pk = 1)

Name:

Title: ▾

Nationality:

Modifica di oggetti esistenti nel DB

```
#file books/views.py
```

```
...
```

```
def modifyAuth(request, a_id):
```

```
    if request.method == 'GET':
```

```
        ...
```

```
    else: # POST request
```

```
        a = Author.objects.get(pk=a_id)
```

```
        form = AuthorForm(request.POST, instance=a)
```

```
        form.save()
```

```
        return HttpResponseRedirect('Author Modified')
```

Salvo i cambiamenti
sul DB

La view prende in ingresso l'id dell'oggetto
da modificare (es. URL “/modifyAuth/1/”)

Recupero l'oggetto
Da modificare dal DB

Modifico l'Author
esistente con i dati inseriti
nella richiesta POST

Verifichiamo
<http://127.0.0.1:8000/books/modifyAuth/1/>

Il metodo save()

- Il metodo **form.save()** effettua anche la validazione dei dati inseriti – controllo sui vincoli
 - Comprende una chiamata a **form.is_valid()**
 - Es. Se non inserito un campo necessario → ripresenta la form
- Save() accetta un **argomento opzionale booleano (commit)** che è a **True per default**
- **Se commit = FALSE** restituisce un **oggetto** che **non** è ancora **salvato** su DB, ma è **manipolabile**
 - Usato per cambiare/processare qualche campo prima del salvataggio finale
- Per il salvataggio finale dovrò poi:
 - Chiamare **save()** sull'**oggetto** restituito
 - Chiamare **save_m2m()** sulla **form** per salvare correttamente le relazioni **many-to-many** dell'**oggetto** (*se presenti nel modello*)

Il metodo save() - esempio

Create a form instance with POST data – to insert a new author

```
>>> f = AuthorForm(request.POST)
```

Create, but don't save the new author instance

```
>>> new_author = f.save(commit=False)
```

Modify the author in some way

```
>>> new_author.some_field = 'some_value'
```

Save the new instance

```
>>> new_author.save()
```

Eseguo save()
sull'oggetto tornato da
f.save(commit=False)

Now, save the many-to-many data for the form

```
>>> f.save_m2m()
```

Necessario poiché
Author ha relazioni
Many-to-many

**Upload di user
generated content**

Upload di user generated content

- L'**upload di contenuti generati da utente** si gestisce agilmente attraverso campi speciali delle **form**
 - **FileField**
 - **ImageField**
- *Nota: **ImageField** richiede l'installazione della **Python Image Library (PIL)** sul sistema*
- Utilizzo di form per upload di file - campo **FileField**

```
#file forms.py (o models.py)
```

```
from django import forms
```

```
class UploadFileForm(forms.Form):
```

```
    title = forms.CharField(max_length=50)
```

```
    file = forms.FileField( )
```

Creazione di una form ad hoc per caricare un file

NOTA: i file caricati in upload (ImageField e FileField) vengono **gestiti separatamente** dai dati "normali" degli altri campi

Upload di user generated content

- Nella view, gli upload di file/immagini vengono recuperati attraverso **request.FILES** (anziché **request.POST**)
 - Dictionary Python: request.FILES['file']
 - Request.FILES contiene una coppia chiave-valore per ogni campo FileField e ImageField contenuto nella form
- Per funzionare correttamente, la form HTML nel **template** deve avere il parametro **enctype** (encoded type)
<form **enctype="multipart/form-data"** method="post">
altrimenti il dictionary request.FILES sarà vuoto
- Per recuperare la form nella views – gestione richiesta POST:
form = UploadFileForm(request.POST, request.FILES)

Nome del campo
FileField della form

Form bound sui dati
Inseriti da utente

Dati nei campi 'normali':
es. 'title'

Dati nei campi ImageField
e FileField: es. 'file'

Uso di UploadFormFile in una view

```
from django.http import HttpResponseRedirect
from django.shortcuts import render_to_response
from .forms import UploadFileForm
```

```
# Ipotetica funzione che gestisce un uploaded file
from somewhere import handle_uploaded_file
```

File views.py

Funzione per la gestione dei file

```
def upload(request):
```

```
    if request.method == 'POST':
```

```
        form = UploadFileForm(request.POST, request.FILES)
```

```
        if form.is_valid():
```

```
            handle_uploaded_file(request.FILES['file'])
```

```
            return HttpResponseRedirect('/success/url/')
```

```
    else: #caso GET
```

```
        form = UploadFileForm()
```

```
        return render(request, 'uploadForm.html', {'form': form})
```

Recupero la form di upload
bounded ai dati inseriti

Validazione

Recupero file

Unbounded form
per inserimento
dati utente

Inseriamo upload in /books/

#file /books/forms.py

```
from django.forms import forms
class UploadFileForm(forms.Form):
    title = forms.CharField(max_length=50)
    file = forms.FileField()
```

#file /books/urls.py

```
# ex: /books/upload/
path('upload/', views.upload, name='upload'),
```

Mapping di /books/upload/
sulla view upload()

#file /books/templates/books/uploadForm.html

```
<form enctype="multipart/form-data" method="post">
{% csrf_token %}
{{ form.as_p }}
<input type="submit" value="Submit" />
</form>
```

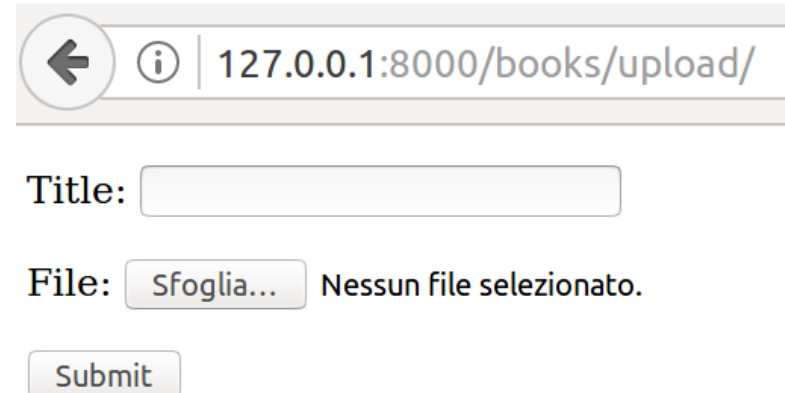
Template uploadForm.html
visualizzato dalla view upload()
da cui riceve la variabile form
in ingresso

Inseriamo upload in /books/

```
#file /books/views.py
from .forms import UploadFileForm

def upload(request):
    if request.method == 'POST':
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            #handle_uploaded_file(request.FILES['file'])
            return HttpResponseRedirect('Successful update')
        return HttpResponseRedirect('Data Not Valid')
    else: #caso GET
        form = UploadFileForm()
        return render(request,
            'books/uploadForm.html', {'form': form})
```

Verifichiamo il funzionamento!
Widget di upload file



The screenshot shows a web browser window with the address bar displaying "127.0.0.1:8000/books/upload/". Below the address bar, there is a form with the following elements:

- A "Title:" label followed by a text input field.
- A "File:" label followed by a file selection button labeled "Sfoglia..." and the text "Nessun file selezionato."
- A "Submit" button at the bottom.

Gestione file dopo l'upload

- **Metodi per gestire il file recuperato**
- **f=request.FILES['file']** restituisce un oggetto di tipo **UploadedFile**
 - Wrapper attorno al file caricato
 - Attributi e metodi: `name`, `size`, `read()`, `chunks()`
- Lettura del file: `read()`, `chunks()`
 - **f.read()** legge il file tutto in una volta e lo posiziona in RAM
 - **f.chunks()** restituisce un generatore che ritorna pezzi del file
- **f.multiple_chunks()**: torna `True` se il file è abbastanza grande da richiedere la lettura in più chunks
- Se **f.multiple_chunk()** ritorna **True** è bene usare **chunks()** al posto di **read()** per non avere problemi di memoria RAM

Gestione file dopo l'upload

- **Procedura comune per il salvataggio di un file:**

```
#file /books/views.py
```

```
def handle_uploaded_file(f):
```

f = request.FILES['file']

```
    with open('name_file', 'wb+') as destination:
```

```
        for chunk in f.chunks():
```

```
            destination.write(chunk)
```

Writing and reading (w+)
in binary format (b)

Dove viene messo il file name_file?
In mysite (BASE_DIR) - Verifichiamo

Costrutto Python

```
>>> with open('namefile','r') as f
```

```
...     read_data = f.read()
```

```
>>> f.closed
```

Molto più compatto del
blocco try-finally

- Preferibile per gestire file: il file viene chiuso correttamente anche nel caso venga sollevata una eccezione

Modelli con campi File/Image

- Caso tipico: il file o l'immagine sono associati ad un model
 - Es. Foto del profilo utente, di un prodotto su sito di aste online, ...
- Il **model** sarà dotato di un campo **FileField** o **ImageField**

#file models.py

```
class ModelWithFileField(models.Model):
```

```
    file_field = models.FileField(...)
```

Vedremo più tardi i
parametri

- Possiamo usare la stessa **Form di upload** generata prima per permettere all'utente di caricare il file

```
class UploadFileForm(forms.Form):
```

```
    title = forms.CharField(max_length=50)
```

```
    file = forms.FileField()
```

Campo file per upload

Form non derivata da model

Si può **assegnare direttamente** l'oggetto File recuperato da `request.FILES` al **campo FileField dell'oggetto ModelWithFileField**

```
from .forms import UploadFileForm
from .models import ModelWithFileField

def upload_file(request):
    if request.method == 'POST':
        form = UploadFileForm(request.POST, request.FILES)
        if form.is_valid():
            instance = ModelWithFileField(file_field=request.FILES['file'])
            instance.save()
            return HttpResponseRedirect('/success/url/')
        else: #GET
            form = UploadFileForm()
            return render(request, 'uploadForm.html', {'form': form})
```

Importo la form per l'upload e il modello con FileField

Recupero i dati (form bound con dati inseriti)

Validazione

Salvo nel DB

Assegno il file al campo dell'oggetto creato

Form unbound- non legata a modelli

Upload con form generate da model

- **Uso di form generata da un model con campo `FileField`**

```
class ModelFormWithFileField(ModelForm):
```

```
    class Meta:
```

```
        model = ModelWithFileField
```

- `#File views.py`

```
from .forms import ModelFormWithFileField
```

```
def upload_file(request):
```

```
    if request.method == 'POST':
```

```
        form = ModelFormWithFileField(request.POST, request.FILES)
```

```
        if form.is_valid():
```

```
            form.save() # object is saved
```

```
            return HttpResponseRedirect('/success/url/')
```

```
else: #GET
```

```
    form = ModelFormWithFileField()
```

```
    return render(request, 'upload.html', {'form': form})
```

Importo la form legata
al modello con `FileField`

Recupero i dati (form
bound con dati inseriti)

Stavolta basta fare una semplice
`save()`: salva il nuovo oggetto nel DB

**MA dove viene salvato
fisicamente il file?**

Dove sono salvati gli upload

- Indicazione nel campo **FileField** o **ImageField** del model
- **Parametro upload_to** che specifica la **directory del file system** in cui **deve essere salvato il file**
 - **upload_to = path relativo nel file system locale** che verrà aggiunto al path specificato in **MEDIA_ROOT**
- **MEDIA_ROOT** in file settings.py
 - Specifica il **percorso assoluto della directory** dove saranno salvati i **file caricati da utente**
 - `MEDIA_ROOT = os.path.join(BASE_DIR, '/media/')`
- Es. `file_field = models.FileField(upload_to="dir_upload/")`
 - I file caricati verranno salvati in:
`/path-to-BASE_DIR/media/dir_upload/`