

# Django Forms

Forms, ModelForms & custom validation

# Riassunto

- Abbiamo visto come sfruttare l'intero design pattern di Django
  - Models (Database relazionali tramite ORM + migrations)
  - views, intese come funzioni e classi
  - e Templates, per il lato presentazione

# Si può migliorare?

Abbiamo sicuramente capito come le CBV ci permettano di risparmiare molto codice nella scrittura della logica della nostra applicazione.

Inoltre, le CBV ci danno in regalo la variabile di contesto “form”, che ci permette di evitare di scrivere il tediosissimo boiler-plate code tipico di HTML e dei suoi `<form>`

# Problemi?

No.

**Se** mi il mio obiettivo è operare metodi CRUD su una tabella.

**Ma** per esempio:

Quando le tabelle diventano due, o quando voglio operare una List view su un form di ricerca con criteri particolari?

Quando voglio esplicitare condizioni e vincoli aggiuntivi per le entry della mia tabella, troppo astrusi per essere codificati nei models?



# Form e validazione personalizzata

Per quello che sappiamo ora, non si scappa...

Non possiamo affidarci alla variabile “form” data in regalo da Django nelle sue CBV di base.

Si pensi all’esempio di Studenti/Insegnamento: in particolare alla ricerca di studenti per nome e/o cognome.

## cerca\_studenti.html

```
{% block content %}

    <center>

    <h1>Cerca studenti</h1>

    <form method="post" action="/iscrizioni/cercastudente/">{% csrf_token %}

        <label for="name">Name:</label><br>

        <input type="text" id="name" name="name" ><br>

        <label for="surname">Surname:</label><br>

        <input type="text" id="surname" name="surname" ><br>

        <input type="submit" value="Cerca">

    </form>

    </center>

{% endblock %}
```

# Forms in DTL

Abbiamo dovuto codificare campo per campo in HTML\DTL ciò che ci interessa.

Abbiamo dovuto esplicitare la logica (boiler-plate) del “se mi arriva un get, rendo un form, se mi arriva un POST leggo gli attributi e mi regolo di conseguenza...”

**DEVE** esserci un modo più intelligente di svolgere queste operazioni in Django.

Mi basta un modo per definire tramite classi/oggetti di python il **mio** form e non quello dato in regalo da Django.

# Forms in Django

Esiste la classe Form in “from django import forms”

Mi basta estendere quella classe per definire due cose importanti:

- 1) Campi editabili (e.g. di ricerca) personalizzati
- 2) Aggiungere condizioni di validazione aggiuntivi a tali campi.

Essendo una classe, segue le regole delle CBV in ottica di riutilizzo/estensibilità.



# Applicazione di esempio

La trovate nel git in

*...django/mysite\_frameCBV/...*

Riprende l'esempio che trovate in ...django/mysite/...

Con queste differenze:

- 1) Progetto “my\_site” con app “polls”, gestito con CBV
- 2) Modelli/tabelle per Question e Choice. Choice ha ora il campo booleano “is\_correct”
- 3) Avremo validazioni di form e ricerche personalizzate tramite i Django Forms.

# Models in polls/models.py

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

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

    def __str__(self):
        return self.question_text

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

    def __str__(self):
        return self.choice_text
```

# init\_db & erase\_db (initcmds.py, funzioni chiamate in urls.py)

La funzione init\_db cambia: questo in accordo con l'aggiunta dell'attributo is\_correct della tabella Choice.  
Dal file static/questions/questions.txt

```
{"response_code":0,"results":[{"category":"Science: Computers","type":"multiple","difficulty":"easy","question":"Which company was established on April 1st, 1976 by Steve Jobs, Steve Wozniak and Ronald Wayne?","correct_answer":"Apple","incorrect_answers":["Microsoft","Atari","Commodore"]},{category":"Science: Computers","type":"multiple","difficulty":"easy","question":"When Gmail first launched, how much storage did it provide for your email?","correct_answer":"1GB","incorrect_answers":["512MB","5GB","Unlimited"]}...]
```

# initcmds.py, init\_db function

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

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

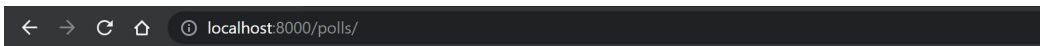
# Possiamo cominciare...

... con un paio di list views (tramite CBV):

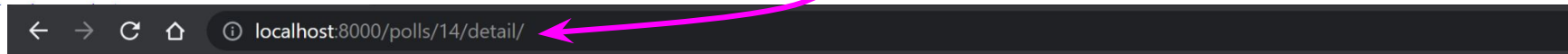
- 1) Home di polls: ListView: le 20 più recenti domande
- 2) /pk/detail/: DetailView: Info sulla domanda univocamente identificata con la sua primary key, unitamente alle sue scelte (e relativi voti).

NB: dalla home, ogni entry listata deve puntare ad un link al sua pagina “detail”

# Index & Detail:



- [What does RAID stand for?](#)
- [Which of the following is the oldest of these computers by release date?](#)
- [On Twitter, what was the original character limit for a Tweet?](#)
- [In the programming language Java, which of these keywords would you put on a variable to make sure it doesn't get modified?](#)
- [Which company was established on April 1st, 1976 by Steve Jobs, Steve Wozniak and Ronald Wayne?](#)
- [Prova](#)
- [How long is an IPv6 address?](#)
- [What is the name of the process that sends one qubit of information using two bits of classical information?](#)
- [In CSS, which of these values CANNOT be used with the "position" property?](#)
- [Which of the following languages is used as a scripting language in the Unity 3D game engine?](#)
- [When Gmail first launched, how much storage did it provide for your email?](#)
- [According to DeMorgan's Theorem, the Boolean expression \(AB\)' is equivalent to:](#)
- [Which of these is not a key value of Agile software development?](#)
- [What is the correct term for the metal object in between the CPU and the CPU fan within a computer system?](#)
- [What does the Prt Sc button do?](#)
- [Moore's law originally stated that the number of transistors on a microprocessor chip would double every...](#)
- [What is the number of keys on a standard Windows Keyboard?](#)
- [This mobile OS held the largest market share in 2012.](#)
- [In programming, the ternary operator is mostly defined with what symbol\(s\)?](#)
- [All of the following programs are classified as raster graphics editors EXCEPT:](#)



**All of the following programs are classified as raster graphics editors EXCEPT:**

- Inkscape 0 votes
- GIMP 1 votes
- Paint.NET 2 votes
- Adobe Photoshop 3 votes

# Le CBV in polls/views.py

```
class IndexViewList(ListView):  
    model = Question  
    template_name = 'polls/index.html'  
  
    def get_queryset(self):  
        return self.model.objects.order_by('-pub_date')[:20]  
  
class DetailQuestion(DetailView):  
    model = Question  
    template_name = "polls/detail.html"
```

# Gli urls in polls/urls.py

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

# I template

Ancora: niente di più di quanto abbiamo già visto...



# Un Form di ricerca

Abbiamo una cinquantina di domande.

Su index, ne vediamo solo 20.

Sarebbe utile poter cercare per esempio un **argomento**: da questa ricerca voglio ottenere le domande che “hanno a che fare” con l’argomento specificato.

Pensiamo un attimo a cosa abbiamo bisogno.

- 1) Una search string per la mia **keyword** (l’argomento)
- 2) **Dove** lo cerco? (Question o Choice)

# Implementiamo il Form di ricerca

Tipicamente si usa creare un file apposito che raccoglie i Form personalizzati della mia applicazione. Lo chiameremo **forms.py**, e come view, model, url etc... può afferire all'intero progetto o alla specifica app.

Un form personalizzato **estende** `django.forms.Form`.

Similmente ai **model**, negli **attributi** andremo a definire i campi editabili che l'utente userà tramite browser.

# Campi editabili di un Form

In `django.forms.(fields|widgets).py`



sono presenti i possibili campi con cui popolare il form:

```
__all__ = (
    "Field",
    "CharField",
    "IntegerField",
    "DateField",
    "TimeField",
    "DateTimeField",
    "DurationField",
    "RegexField",
    "EmailField",
    "FileField",
    "ImageField",
    "URLField",
    "BooleanField",
    "NullBooleanField",
    "ChoiceField",
    "MultipleChoiceField",
    "ComboField",
    "MultiValueField",
    "FloatField",
    "DecimalField",
    "SplitDateTimeField",
    "GenericIPAddressField",
    "FilePathField",
    "JSONField",
    "SlugField",
    "TypedChoiceField",
    "TypedMultipleChoiceField",
    "UUIDField",
)
```

A ciascun campo associamo un widget, ossia un componente grafico:

```
__all__ = (
    "Media",
    "MediaDefiningClass",
    "Widget",
    "TextInput",
    "NumberInput",
    "EmailInput",
    "URLInput",
    "PasswordInput",
    "HiddenInput",
    "MultipleHiddenInput",
    "FileInput",
    "ClearableFileInput",
    "Textarea",
    "DateInput",
    "DateTimeInput",
    "TimeInput",
    "CheckboxInput",
    "Select",
    "NullBooleanSelect",
    "SelectMultiple",
    "RadioSelect",
    "CheckboxSelectMultiple",
    "MultiWidget",
    "SplitDateTimeWidget",
    "SplitHiddenDateTimeWidget",
    "SelectDateWidget",
)
```

<https://docs.djangoproject.com/en/4.0/ref/forms/fields/>

# Cosa scegliamo?

```
_all_ = (  
    "Field",  
    "CharField",  
    "IntegerField",  
    "DateField",  
    "TimeField",  
    "DateTimeField",  
    "DurationField",  
    "RegexField",  
    "EmailField",  
    "FileField",  
    "ImageField",  
    "URLField",  
    "BooleanField",  
    "NullBooleanField",  
    "ChoiceField",  
    "MultipleChoiceField",  
    "ComboField",  
    "MultiValueField",  
    "FloatField",  
    "DecimalField",  
    "SplitDateTimeField",  
    "GenericIPAddressField",  
    "FilePathField",  
    "JSONField",  
    "SlugField",  
    "TypedChoiceField",  
    "TypedMultipleChoiceField",  
    "UUIDField",  
)
```

La mia search string per la keyword

Per scegliere dove cercare

# E i widget?

Niente colpi di testa per ora, prendiamo quelli di default associati ai field scelti

Dalla guida di Django:

## CharField

*class* **CharField**(\*\*kwargs)

- Default widget: **TextInput**
- Empty value: Whatever you've given as **empty\_value**.
- Normalizes to: A string.
- Uses **MaxLengthValidator** and **MinLengthValidator** if **max\_length** and **min\_length** are provided. Otherwise, all inputs are valid.
- Error message keys: **required, max\_length, min\_length**

Has four optional arguments for validation:

**max\_length**

**min\_length**

If provided, these arguments ensure that the string is at most or at least the given length.

**strip**

If **True** (default), the value will be stripped of leading and trailing whitespace.

**empty\_value**

The value to use to represent "empty". Defaults to an empty string.

## ChoiceField

*class* **ChoiceField**(\*\*kwargs)

- Default widget: **Select**
- Empty value: ' ' (an empty string)
- Normalizes to: A string.
- Validates that the given value exists in the list of choices.
- Error message keys: **required, invalid\_choice**

# In polls/forms.py (occorre crearlo)

```
from django import forms
```

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

```
    CHOICE_LIST = [("Questions","Search in Questions"),  
                   ("Choices","Search in Choices")]
```

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

# Come raggiungiamo quel form?

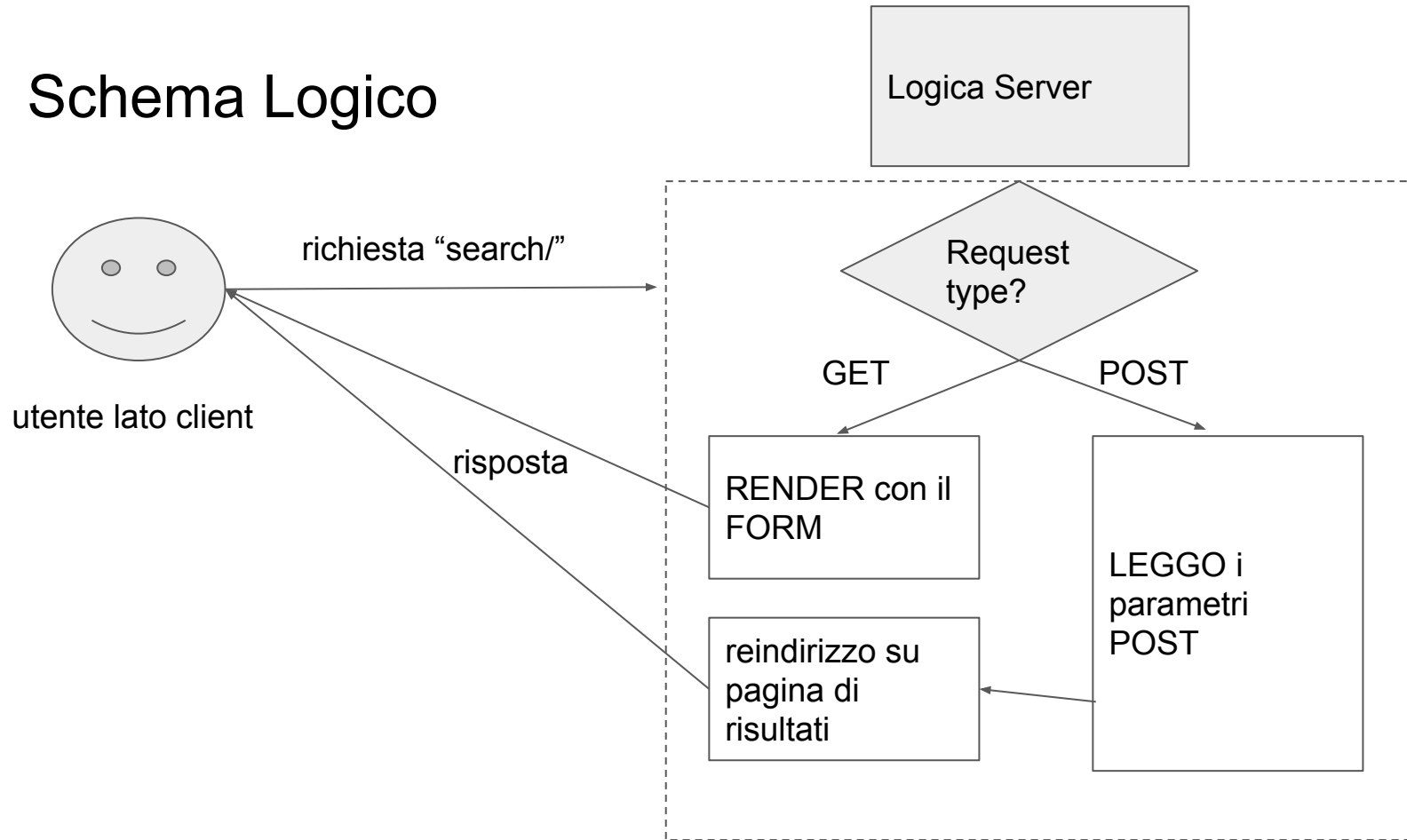
In polls/urls.py:

```
path("search/", search, name="search"),
```

```
path("searchresults/<str:sstring>/<str:where>/", SearchResultsList.as_view(), name="searchresults")
```

in search, risponde una function view. Assumendo che i dati riempiti dall'utente tramite form usino il metodo POST, possiamo raggiungere la prima volta l'url "search/" tramite richiesta GET. Al click del pulsante submit, i dati inseriti (nei campi definiti dal SearchForm) re-indirizzeranno sul secondo url, i cui parametri sono compilati in funzione di request.POST

# Schema Logico





# In polls/views.py

```
from django.views.generic.list import ListView
from django.shortcuts import get_object_or_404, redirect, render
from .forms import *

def search(request):

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

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

# Fermiamoci un attimo

Prima di vedere dove porta il redirect, facciamo alcune osservazioni.

Se la richiesta è GET, instanziamo senza parametri un oggetto di tipo `SearchForm`, ossia il nostro form personalizzato. Semplicemente, lo passeremo come variabile di contesto al nostro template (`polls/template/polls/searchpage.html`):

```
{% extends "base.html" %}
{% block title %} Search Page Form {% endblock %}
{% block content %}

<h1>Search Page</h1>

<form action="{% url 'polls:search' %}" method="POST">  {% csrf_token %}

    {{form.as_p}}

<input type="submit" value="Search"> </form>

{% endblock %}
```

# La variabile “form”

Non è più data in regalo da una CBV di Django. E' nostra, e l'abbiamo pure creata noi. Segue comunque le stesse regole di formattazione che abbiamo visto con i form associati alle CBV che abbiamo visto precedentemente. In particolare:

- 1) `as_p`, `as_table` e `as_ul` funzionano ancora.
- 2) Nel form, usiamo sia il metodo POST che il token csrf.
- 3) Il form lo dobbiamo comunque aprire e chiudere noi:
  - a) `<form method=... action=...>`
  - b) Inserire il pulsante, l'elemento input di tipo “submit”
  - c) `</form>`

# Su localhost:8000/polls/search/

## Search Page

Search String:

Search Where?

```
class SearchForm(forms.Form):  
  
    CHOICE_LIST = [("Questions", "Search in Questions"), ("Choices", "Search in Choices")]  
  
    search_string = forms.CharField(label="Search String", max_length=100, min_length=3, required=True)  
    search_where = forms.ChoiceField(label="Search Where?", required=True, choices=CHOICE_LIST)
```

```
<form action="/polls/search/" method="POST">  
  <input type="hidden" name="csrfmiddlewaretoken" value="IgBYqSV03VZO  
uCHxuJKeQKfhaLs3iOQLiwJFbWbaz9HGttEH8VRnV3QAteGpoPv6">  
  <p>  
    <label for="id_search_string">Search String:</label>  
    <input type="text" name="search_string" maxlength="100"  
    minlength="3" required id="id_search_string">  
  </p>  
  <p>  
    <label for="id_search_where">Search Where?</label>  
    <select name="search_where" id="id_search_where"> == $0  
      <option value="Questions">Search in Questions</option>  
      <option value="Choices">Search in Choices</option>  
    </select>  
  </p>  
  <input type="submit" value="Search">  
</form>
```

# Osservazioni

Per ogni elemento creato nel Form personalizzato

- è associata una label. Quest'ultima verrà creata come elemento HTML aggiuntivo.
- appare in forms.py come una variabile. Il nome della variabile riempie il campo "name" del rispettivo elemento HTML. Ottenibile poi tramite request.POST
- hanno ulteriori parametri (e.g. min/max length), tradotti appositamente in attributi di elementi HTML

# Torniamo su polls/views.py

```
from django.views.generic.list import ListView
from django.shortcuts import get_object_or_404, redirect, render
from .forms import *
```

```
def search(request):
```

```
    if request.method == "POST":
        form = SearchForm(request.POST)
        if form.is_valid():
            sstring = form.cleaned_data.get("search_string")
            where = form.cleaned_data.get("search_where")
            return redirect("polls:searchresults", sstring, where)
```

```
    else:
```

```
        form = SearchForm()
```

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

# Se il metodo è POST...

In request.POST abbiamo i dati che ci servono. Tutti quelli con attributo “name” specificato.

**Però**, non accediamo direttamente a quel dizionario.

“Creiamo” un SearchForm dandogli come parametro in ingresso il dizionario. Il sistema, anzichè istanziare un form vuoto, legge i dati passati e li **valida**. In maniera tale che con le istruzioni `form.cleaned_data.get(...)` ci permette di avere un input “sanitizzato”. Vedremo in seguito come sarà possibile introdurre in un form regole di validazione personalizzate

# Redirezione

con la shortcut “redirect” siamo in grado di dirottare i campi letti verso l'url resolver. In particolare andremo a chiamare la view risolta tramite “polls:searchresults” che ammette due parametri tramite url. Una stringa per la keyword ed una stringa che identifica la table su cui vogliamo fare la query.

Dato che mi aspetto una lista di risultati, posso creare una ListView. Tenendo conto però che dovrò fare override del metodo `get_queryset` dato che la tabella coinvolta è decisa a runtime.



# La CBV dei risultati

```
class SearchResultsList(ListView):

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

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

        if "Question" in where:
            qq = Question.objects.filter(question_text__icontains=sstring)
        else:
            qc = Choice.objects.filter(choice_text__icontains=sstring)
            qq = Question.objects.none()
            for c in qc:
                qq |= Question.objects.filter(pk=c.question_id)

    return qq
```

# Il template

Niente di particolare.

In quanto eredita da ListView, sappiamo che esiste una variabile di contesto `object_list`, che conterrà oggetti di tipo **Question** che noi siamo in grado di visualizzare a livello di DTL.

localhost:8000/polls/search/

# Search Page

Search String:

Search Where?

Search

localhost:8000/polls/search/

# Search Page

Search String:

Search Where?

Search

localhost:8000/polls/searchresults/apple/Questions/

- [While Apple was formed in California, in which western state was Microsoft founded?](#)
  - Arizona
  - New Mexico
  - Washington
  - Colorado
- [Which of these people was NOT a founder of Apple Inc?](#)
  - Ronald Wayne
  - Jonathan Ive
  - Steve Wozniak
  - Steve Jobs
- [Approximately how many Apple I personal computers were created?](#)
  - 100
  - 1000
  - 500
  - 200

localhost:8000/polls/searchresults/apple/Choices/

- [Which company was established on April 1st, 1976 by Steve Jobs, Steve Wozniak and Ronald Wayne?](#)
  - Microsoft
  - Apple
  - Commodore
  - Atari
- [Which of the following is the oldest of these computers by release date?](#)
  - Commodore 64
  - Apple 3
  - TRS-80
  - ZX Spectrum

# Integrazione tra Models e Forms

Così come abbiamo visto per le CBV, possiamo associare un Form ad una particolare tabella, se essa è stata definita come oggetto python tra i nostri model.

Cosa ci permette di fare l'integrazione tra Models & Forms?

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

# ModelChoiceField

Si faccia una view che permetta di votare e quindi rispondere ad una question.

La question è selezionata tramite pk. Per comodità si renda raggiungibile la domanda tramite link url per ciascun elemento della IndexView e/o pagina di searchresults di polls.

Il voto della domanda avviene tramite Form. Esso deve avere un campo a scelta multipla in cui, dalla domanda, l'utente seleziona una delle quattro risposte.

Al submit del form, il sistema redireziona in una DetailView apposita, la quale:

- Informa l'utente se la sua risposta è corretta o meno,
- Incrementa il campo "votes" della risposta.

## II Form

```
from django import forms
from django.shortcuts import get_object_or_404
from .models import *
```

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

```
    answer = forms.ModelChoiceField(queryset=None, required=True, label="Select your answer!")
```

```
    def __init__(self, pk, *args, **kwargs):
```

```
        super().__init__(*args, **kwargs)
```

```
        q = get_object_or_404(Question, pk=pk)
```

```
        self.fields['answer'].queryset = q.choices.all()
```

## Il campo “answer”

Somiglia al campo “choiceField” visto in precedenza.

Ma questa volta è legato ad un modello: infatti le possibili scelte non sono hardcoded da una variabile come abbiamo visto nel form precedente.

```
answer = forms.ModelChoiceField(queryset=None, ....)
```

Bensì, sono prese da un queryset, quindi da un modello. **Alla sua dichiarazione, tale campo è posto a None.**

Per riempire l'attributo queryset di answer abbiamo fatto override del metodo `__init__`, il quale rispetto al padre “forms.Form”, prende ora in ingresso anche la pk della Question a cui vogliamo rispondere





# Osservazioni

Rispetto all'esempio precedente abbiamo una sola sostanziale differenza:

Il modo in cui abbiamo istanziato `VoteForm` nei due casi (GET e POST).

In entrambi i casi abbiamo dovuto esplicitare il parametro costruttivo aggiuntivo che abbiamo chiamato “pk”.

Serve per “riempire” le scelte del nostro `ModelChoiceField`.

Per il resto, redirectione su una CBV di tipo `DetailView` che userà il parametro `pk` per mettere nella variabile di contesto “object” la question corrispondente.

Su questa View, faremo override sul metodo `get_context_data` per aggiungere le informazioni richieste.

# Redirezione (votecasted/<pk>/<str:answer>/)

```
class VoteCastedDetail(DetailView):  
    model = Question  
    template_name = "polls/votecasted.html"  
  
    def get_context_data(self, **kwargs):  
        ctx = super().get_context_data(**kwargs)  
        answer = self.request.resolver_match.kwargs["answer"]  
        ctx["answer"] = answer  
        correct = ctx["object"].choices.all().get(is_correct=True)  
        if answer in correct.choice_text:  
            ctx["message"] = "Right Answer!"  
        else:  
            ctx["message"] = "Wrong Answer! " + " The right answer was " + str(correct.choice_text)  
  
        try:  
            c = ctx["object"].choices.all().get(choice_text=answer)  
            c.votes += 1  
            c.save()  
        except Exception as e:  
            print("Impossible to update vote value " + str(e))  
  
    return ctx
```

- [Moore's law originally stated that the number of transistors on a microprocessor chip would double every...](#)
  - Four Years
  - Year
  - Two Years
  - Eight Years
- [On which computer hardware device is the BIOS chip located?](#)
  - Central Processing Unit
  - Motherboard
  - Graphics Processing Unit
  - Hard Disk Drive

## On which computer hardware device is the BIOS chip located?

- Central Processing Unit 0 votes
- Motherboard 2 votes
- Graphics Processing Unit 2 votes
- Hard Disk Drive 3 votes

[Give an answer!](#)

## Vote Page

On which computer hardware device is the BIOS chip located?

Select your answer!

## On which computer hardware device is the BIOS chip located?

Your answer was:

Motherboard

Right Answer!

[Homepage](#)

# ModelForm e CBV

Nell'esempio precedente abbiamo legato un campo del Form ad un queryset, quindi ad un modello.

Abbiamo già detto che invece legare l'intero Form ad un Model ci permette di avere controllo aggiuntivo su come l'utente specifica i dati in ingresso nelle pagine web.

In django esistono quindi i ModelForm nello stesso modulo dei form "tradizionali".

Seguono le regole dei form che abbiamo già visto, ma danno la possibilità di creare meta informazioni (classe nested Meta) in cui possiamo specificare a quale tabella sono collegati.

Una volta specificato il model di riferimento e l'attributo fields (nella stessa maniera che abbiamo visto con le CreateView, per esempio), non occorre poi specificare i campi del form...

# Esempio

S'implementino due CBV per la creazione di Question e Choice rispettivamente. Ma si considerino i seguenti vincoli:

- **CreateQuestionForm:**
  - Una domanda non può avere meno di 5 caratteri.
- **CreateChoiceForm:**
  - Non devo poter inserire una choice ad una question che ha già 4 risposte.
  - Esiste una sola risposta corretta su quattro

# In polls/forms.py

```
class CreateQuestionForm(forms.ModelForm):

    description = "Create a new Question!"

    def clean(self):

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

        return self.cleaned_data

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

class CreateChoiceForm(forms.ModelForm):

    description = "Create choices for a question"

    def clean(self):

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

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


        if(choices.count()==4):
            self.add_error("question", "Error: question already has four options")
        elif(choices.count()==3):
            if choices_false.count()==3 and self.cleaned_data["is_correct"] == False:
                self.add_error("is_correct", "Error: exactly one choice must be correct")

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

        return self.cleaned_data

    class Meta:
        model = Choice
        fields = "__all__"
```

# Osservazioni

- Entrambi i form ereditano da `forms.ModelForm` 
- Entrambi i form hanno una variabile chiamata *description*. Non è necessaria.
- Entrambi i form hanno specificato gli attributi *model* e *fields* nella loro Meta classe interna. Similmente a quanto accadeva nelle *CreateViews*
- In entrambi i form abbiamo fatto override del metodo *clean*: questo ci consente di implementare validazione aggiuntiva sugli input dell'utente.
- La validazione standard è già stata eseguita **prima** del metodo *clean*. In particolare i dati pre-validati sono disponibili in *self.cleaned\_data*. Il quale è un dizionario le cui chiavi corrispondono agli attributi (stringa) del model specificato.
- Gli errori che aggiungiamo in *clean*, se scatenati, compariranno nel render del browser e **impediranno la sottomissione dei dati POST**.
  - *self.add\_error("campo interessato", "stringa associata all'errore")*

# CreateQuestionForm.Meta.widgets

CreateQuestionForm ha una particolarità che l'altro ModelForm non ha.

Un parametro in più nelle sue META informazioni.

Tale parametro si chiama `widgets` ed appare come un dizionario a cui nella chiave associamo sottoforma di stringa un attributo del model in questione ad un **widget diverso da quello che userebbe di default**.

In questo caso specifico, andiamo a cambiare il widget di default associato all'attributo *pub\_date*. Vogliamo costringere la data di pubblicazione ad essere scelta da un calendario in formato Giorno/Mese/Anno.



# Le Views in polls/views.py (e rispettivi urls)

```
class CreateQuestionView(CreateView):  
    template_name = "polls/createentry.html"  
    form_class = CreateQuestionForm  
    success_url = reverse_lazy("polls:index")  
  
class CreateChoiceView(CreateView):  
    template_name = "polls/createentry.html"  
    form_class = CreateChoiceForm  
  
    def get_success_url(self):  
        ctx = self.get_context_data()  
        pk = ctx["object"].question.pk  
        return reverse("polls:detail",kwargs={"pk":pk})
```

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

Che differenze avrei avuto se **non** avessi utilizzato un form personalizzato?

## II Template: polls/templates/polls/createentry.html

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

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

{% block content %}

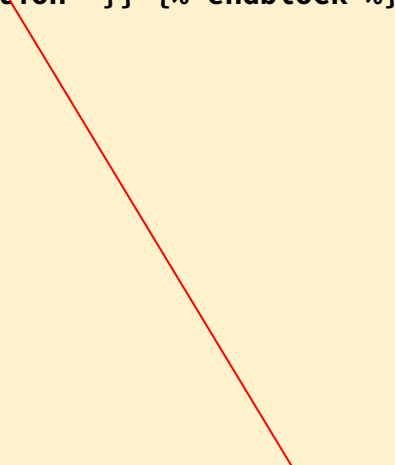
<h1> {{ view.form_class.description }} </h1>

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

    {{form.as_p}}

<input type="submit" value="Create">
</form>

{% endblock %}
```

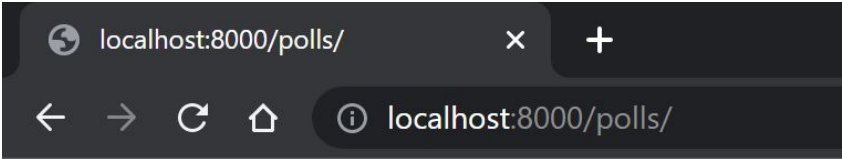


# Create a new Question!

Question text:

Date published:

Create



- [Domanda di Prova](#)

# Create choices for a question

Question:

Choice text:

Votes:

Is correct: ☒

Create

# Domanda di Prova

- Risposta Sbagliata 0 votes
- Risposta Sbagliata 0 votes
- Risposta Sbagliata 0 votes
- Risposta Giusta 0 votes

[Give an answer!](#)

# Create a new Question!

- Error: question text must be at least 5 characters long

Question text:

Date published:  📅

Create

## Create choices for a question

- Error: question already has four options

Question:

Choice text:

Votes:

Is correct: ☐

Create