11 - Forms

Django Forms

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?

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.hml`

```
{% 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 %}
```

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`, rendero 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.

Django ci salva dicendo: "Visto che ti sei divertito a scrivere entry di una tabella come se fossero oggetti, ora puoi farlo anche con i form che normalmente scrivi in html".

I forms come classi

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.

Esercizio: form di ricerca

Esempio

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 la cerco? (Question o Choices)

models.py

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
```

forms.py

Creare un file che raccolga tutti i forms => `forms.py`.
Come `view.py`, `models.py`, `urls.py` etc…, può afferire all'intero progetto o alla specifica app.
Un form personalizzato estende `django.forms.Form`.

Campi editabili di un form (fields) e widgets

Sono:

```
__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",
)
```

In generale ogni field ha associato un widget, ossia un componente grafico. Essi sono:

```
__all__ = (
    "Media",
    "MediaDefiningClass",
    "Widget",
    "TextInput",
    "NumberInput",
    "EmailInput",
    "URLInput",
    "ColorInput",
    "SearchInput",
```

```
    "TelInput",
    "PasswordInput",
    "HiddenInput",
    "MultipleHiddenInput",
    "FileInput",
    "ClearableFileInput",
    "Textarea",
    "DateInput",
    "DateTimeInput",
    "TimeInput",
    "CheckboxInput",
    "Select",
    "NullBooleanSelect",
    "SelectMultiple",
    "RadioSelect",
    "CheckboxSelectMultiple",
    "MultiWidget",
    "SplitDateTimeWidget",
    "SplitHiddenDateTimeWidget",
    "SelectDateWidget",
)
```

Associazioni fields - widgets:

Field	Default widgets
`BooleanField`	`CheckboxInput`
`CharField`	`TextInput`
`ChoiceField`	`Select`
`DateField`	`DateInput`
`DateTimeField`	`DateTimeInput`
`DecimalField`	`NumberInput` when `Field.localize` is False, else `TextInput`
`DurationField`	`TextInput`
`EmailField`	`EmailInput`
`FileField`	`ClearableFileInput`
`FilePathField`	`Select`
`FloatField`	`NumberInput` when `Field.localize` is False, else `TextInput`
`GenericIPAddressField`	`TextInput`
`ImageField`	`ClearableFileInput`
`IntegerField`	`NumberInput` when `Field.localize` is False, else `TextInput`
`JSONField`	`Textarea`
`MultipleChoiceField`	`SelectMultiple`
`NullBooleanField`	`NullBooleanSelect`
`RegexField`	`TextInput`
`SlugField`	`TextInput`

Field	Default widgets
`TimeField`	`TimeInput`
`TypedChoiceField`	`Select`
`TypedMultipleChoiceField`	`SelectMultiple`
`URLField`	`URLInput`
`UUIDField`	`TextInput`

`ComboField`	`TextInput`
`MultiValueField`	`TextInput`
`SplitDateTimeField`	`SplitDateTimeWidget`

`ModelChoiceField`	`Select`
`ModelMultipleChoiceField`	`SelectMultiple`

Scegliamo `CharField` per la search string e `ChoiceField` per scegliere dove cercare (nelle questions o nelle choices).

Creiamo il form di ricerca in `polls/forms.py`:

```
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)
```

urls.py

In `polls/urls.py`:

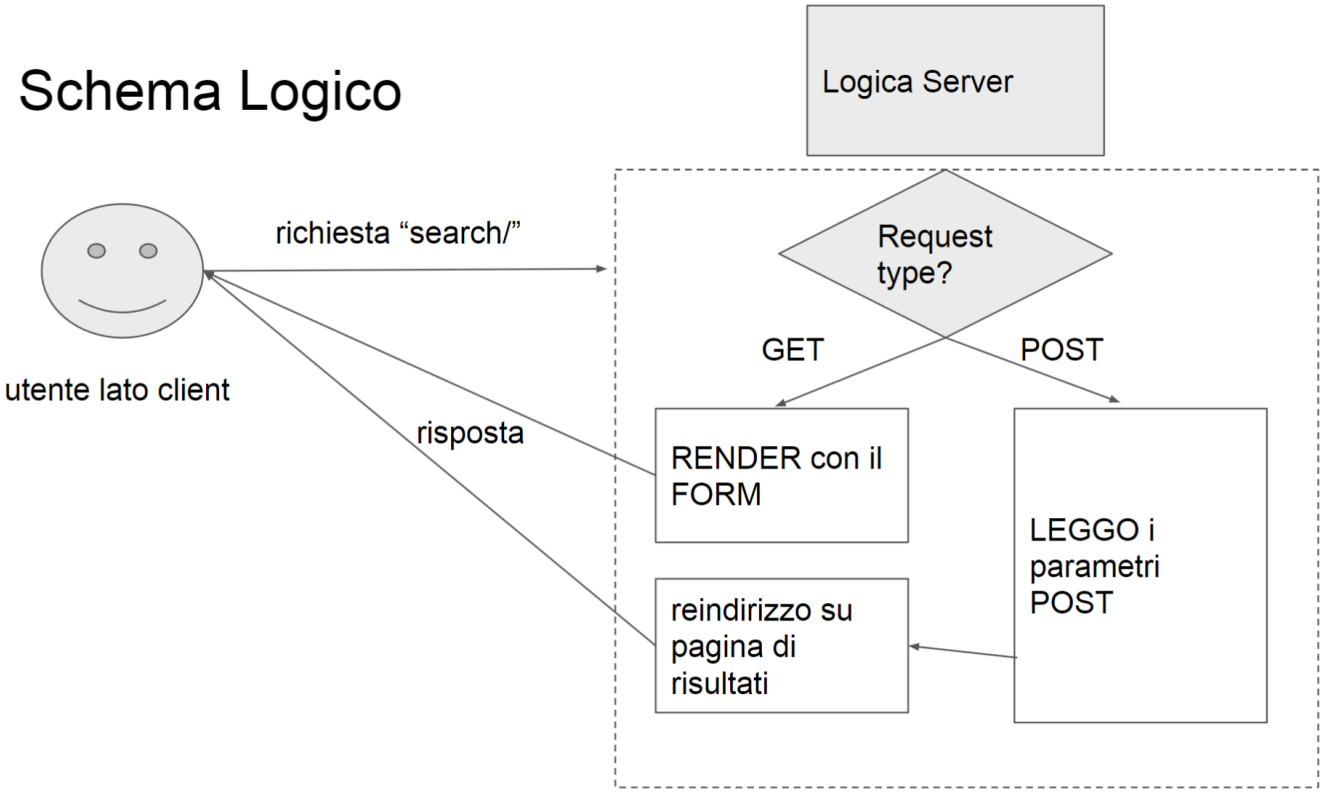
```
path("search/", search, name="search"),
path("searchresults/<str:sstring>/<str:where>/", SearchResultsList.as_view(),
name="searchresults")
```

In `search/` risponde una function view.
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 (assumendo che i dati riempiti dall'utente tramite form usino il

metodo POST).
Questo ragionamento continua in views.py, dove si vede il codice che lo esegue.

views.py

Il ragionamento è lo stesso che viene fatto nel view.py dell'esercizio complesso 10 - Class Based Views > ^5340dc. Solo che qui il codice che lo esegue è scritto molto meglio.



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})
```

Prima di vedere dove porta il redirect osserviamo che:

Se la richiesta è `GET`

entriamo nell'`else` (e vuol dire che siamo appena arrivati sul sito "per la prima volta").
Istanziamo 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):
In `templates/polls/searchpage.html`

```
{% extends "base.html" %}
{% block title %} Search Page Form {% endblock %}
{% block content %}
        <h1>Search Page</h1>
        <form action="{% url 'polls:search' %}" method="POST"> {% csrf_token %}
                {{form.as_p}}
        <input type="submit" value="Search"> </form>
{% endblock %}
```

La variabile `form` non è più data in regalo da una CBV di Django. È 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:
    1. `<form method=... action=...>`
    2. Inserire il pulsante, ovvero l'elemento input di tipo "submit"
    3. `</form>`

Quindi si ottiene:

Su localhost:8000/polls/search/

```
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)
```

Search Page

Search String:

Search Where? Search in Questions

Search

```
<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. mix/max length), tradotti appositamente in attributi di elementi HTML

Se la richiesta è `POST`

entriamo nel primo `if` (e vuol dire che eravamo sul sito, abbiamo compilato il form e abbiamo cliccato il pulsante "submit").
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.
Questa è la view dell'url su cui veniamo ridirezionati (ovvero la view che mostrerà il risultato della richiesta). In `polls/views.py` aggiungiamo:

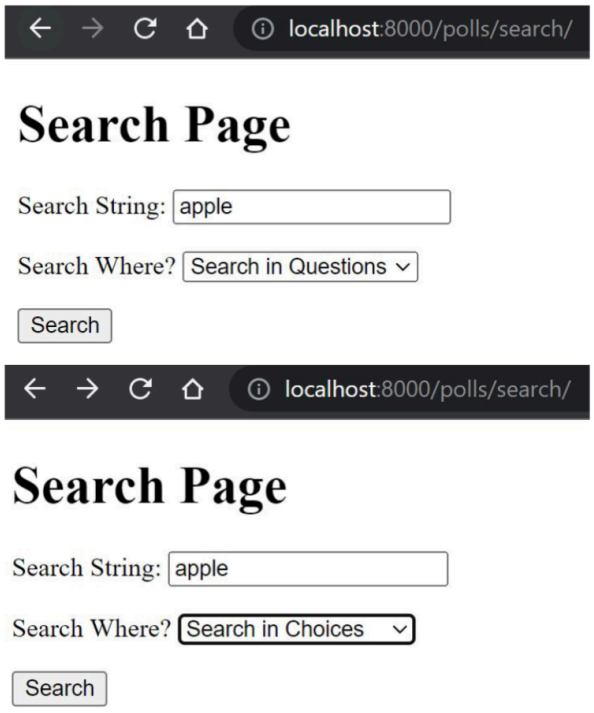
```
class SearchResultsList(ListView):

    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
```

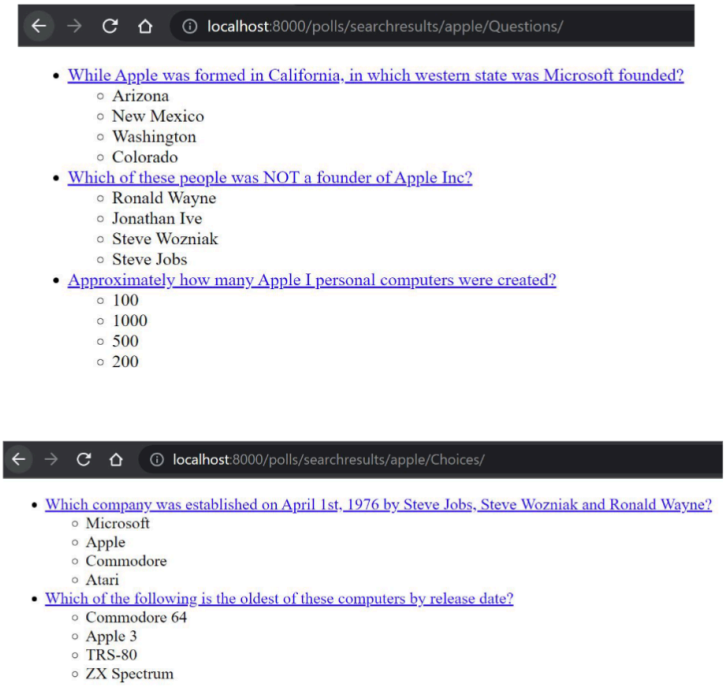
Il template non è niente di particolare.
In quanto eredita da ListView, sappiamo che esiste una variabile di contesto object_list, che conterrà oggetti di tipo Question che noi siamo in grado di visualizzare a livello di DTL.
Esso è `templates/polls/searchresults.html`:

```
{% extends "base.html" %}
{% block title %} Search Results  {% endblock %}
```

```
{% block content %}

        {% if object_list %}
            <ul>
                    {% for q in object_list %}
                            <li><a href="{% url 'polls:detail' q.id %}">{{
q.question_text }}</a></li>
                        <ul>
                                {% for c in q.choices.all %}
                                    <li>{{ c.choice_text }}</li>
                            {% endfor %}
                        </ul>
                    {% endfor %}
            </ul>
        {% else %}
            <p>No results.</p>
        {% endif %}

{% endblock %}
```

Quindi il risultato è:





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.

Voglio creare un form personalizzato, che abbia come scelte multiple non dei valori hardcodati con una variabile (come abbiamo visto prima), ma dei valori presi da un queryset di un modello (ovvero una tabella).
Per fare ciò si usano i Model Form, che sono dei form legati a dei modelli. ⬇️

Esercizio: ModelChoiceField

> **🖊 Risposta ad una question**
>
> 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 (ovvero quando si risponde alla domanda), 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.

Il `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 form è un normalissimo form, non è un Model Form.
Le risposte non le modelliamo più con un ChoiceField, ma con un `ModelChoiceField`. Esso prende in ingresso

- un `queryset`, che inizializziamo a nullo, siccome quello è un attributo che verrà inizializzato chissà quando (forse nelle operazioni di startup del progetto, boh), quindi di default lo metto a nullo;
- `required=True`, rispondere è obbligatorio, ovvero devo scegliere almeno una opzione del ModelChoiceField;

- `label`, quello che ci viene scritto dentro al widget del ModelChoiceField quando non è ancora stato selezionato niente.

Quando è che sicuramente ci serve che il queryset sia inizializzato? Quando istanziamo il form. E quando istanziamo il form chiamiamo `__init__`, che è il costruttore della classe (che rappresenta il form).
Quindi in tale metodo inizializziamo il queryset dal model `Question`.

Nota che si utilizza il metodo `get_object_or_404()`, così in caso di chiave assente, viene dato un 404.

Come faccio ad essere sicuro che quel queryset contenga una domanda? Perché faccio match con la chiave primaria. Poi inizializzo il queryset con le risposte di quella domanda.

Nota poi che utilizziamo la proprietà di riflessività di una classe: essa infatti non è altro che un dizionario le cui chiavi sono gli attributi della classe. Quindi posso chiamare `self.fields["chiave/attributo"]` e andare a cambiare il valore di quella specifica istanza, non del valore di default definito nella classe.

La `view` di voto

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

La view è la stessa identica dell'altra volta. Essa è raggiungibile dall'url `vote/pk/`: nota infatti che la prima volta che arriviamo sull'url della votazione, in cui noi generiamo il form per rispondere, l'unico dato che abbiamo è la primary key della domanda contenuta nell'url.
È sempre lo stesso ragionamento:

1. arriviamo su questo url tramite richiesta GET, quindi entriamo nell'else e generiamo il form con la domanda giusta (troviamo la domanda giusta tramite la pk passata nell'url);
2. compiliamo il form e quando submittiamo, è come se facessimo una richiesta POST sullo stesso url, quindi rieseguiamo la FBV e questa volta entriamo nell'if, dove dopo aver raccolto in forma "sanitizzata" l'input dell'utente, lo redirezioniamo all'url la cui view mostrerà il risultato della risposta.
   Tale view è:

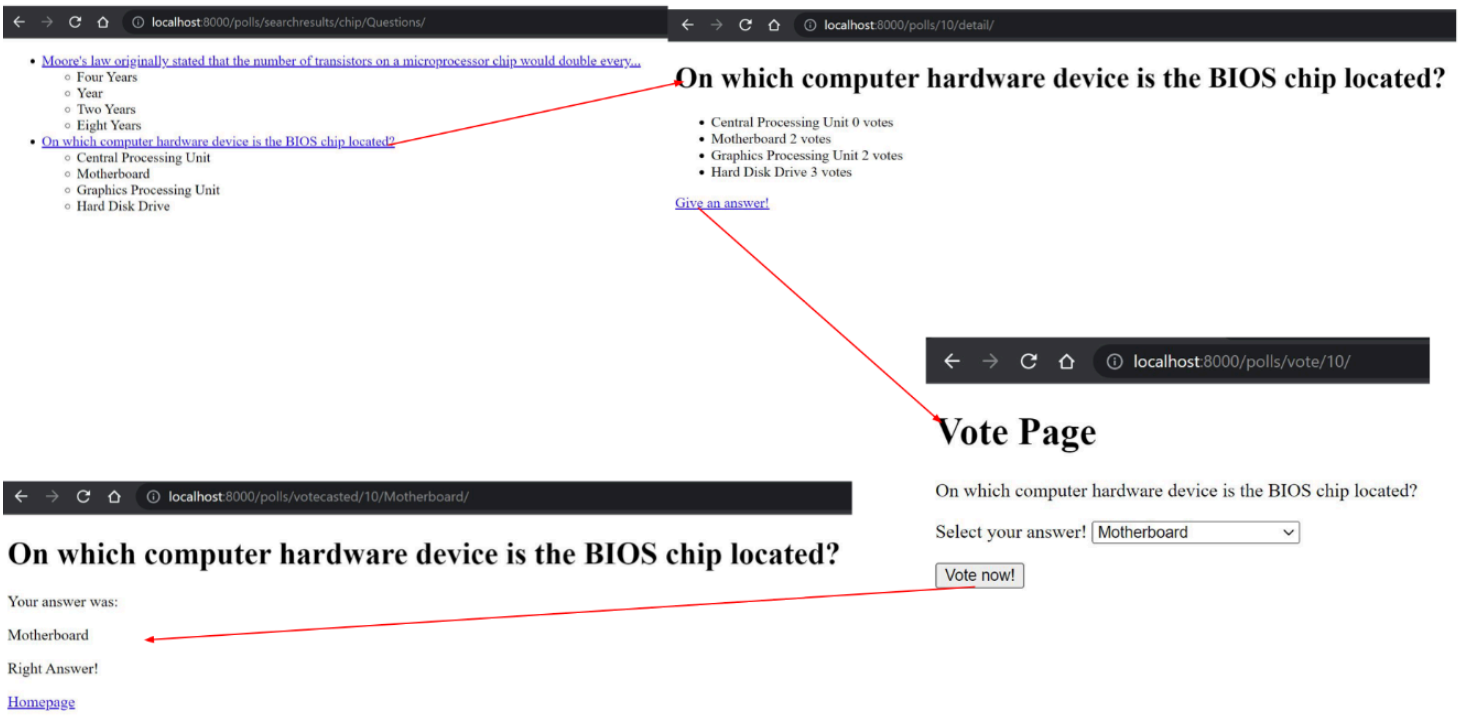
```
class VoteCastedDetail(DetailView):
    model = Question
    template_name = "polls/votecasted.html"
```

```
    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
```

Dato che devo mostrare la risposta ad una domanda, la CBV eredita da DetailView.

Ora, dove metto la logica che conta le risposte giuste?
La potrei mettere dove voglio, perché presto o tardi, tutti i metodi della nostra DetailView vengono chiamati. Ma decido di metterla in `get_context_data()` piuttosto che in qualche altro metodo, così implemento una variabile contatore che poi vado ad usare come variabile di contesto che viene quindi passata all'utente nel template html. In questo modo sto sia implementando la logica che conta le risposte giuste sia fornendo un modo all'utente per capire se la risposta che ha inserito è corretta (perché appunto vedrà il contatore aumentare di 1).
Lo dice a lezione, ma poi non fa così nel codice. Semplicemente aumenta di 1 l'attributo "votes" della choice che ha scelto l'utente.

Il risultato



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

I nostri modelli (cioè le tabelle) non sono difficili, ma potrebbero esserlo le loro regole di validazione.

🖊 Esempi da soddisfare

- `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.

I `forms`

`CreateQuestionForm` in `poll/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'})
        }
```

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.

`CreateChoiceForm` in `poll/forms.py`:

```
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)
        # Se le scelte sono già 4
        if(choices.count()==4):
                # l'utente non può aggiungerne un'altra
            self.add_error("question", "Error: question already has four options")
        # Se le scelte sono 3
        elif(choices.count()==3):
                # Se le scelte sbagliate sono 3 ed è falsa anche quella che sta
inserendo adesso (self è quella che sta istanziando ora)
            if choices_false.count()==3 and self.cleaned_data["is_correct"] == False:
                    # allora è errore, cioè sto imponendo che quella che sta
istanziando adesso sia vera (dato che ne ho già 3 false)
                self.add_error("is_correct", "Error: exactly one choice must be
correct")

                # Se c'è già una risposta corretta ed è corretta anche quella che sta
inserendo adesso
```

```
        if (choices.filter(is_correct=True).count()==1 and
self.cleaned_data["is_correct"] == True):
                #errore, ci può essere solo una risposta vera
            self.add_error("is_correct", "Error: This question already has a correct
answer")

        return self.cleaned_data

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

Osservazioni miste a teoria

- 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 classe interna Meta. Similmente a quanto accadeva nelle CreateViews.
    - Quindi è in questa classe che vado a definire il form a quale model fa riferimento e quali attributi di esso voglio permettere all'utente di cambiare.

  Nota che in tale classe è anche possibile overridare i widget grafici che vengono assegnati di default ai fields.
- 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 tramite la funzione `add_error()`, se scatenati, compariranno nel render del browser e impediranno la sottomissione dei dati POST.
    - `self.add_error("campo interessato","stringa associata all'errore")`
        - la prima stringa è il nome dell'attributo del model su cui l'utente sta cercando di salvare un valore non valido;
        - la seconda è il messaggio di errore vero e proprio.

Le `views` e gli `urls`

In `polls/views.py`:

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

class CreateChoiceView(CreateView):
```

```
    template_name = "polls/createentry.html"
    #template_name = "polls/crispyf/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})
```

Questo forte legame che c'è tra model e forms (appunto i ModelForms), fa si che tutto quel codice dei form che avevamo fatto in modo manuale (prima di aver visto le CBV, oppure per averli personalizzati e non quelli predefiniti di Django), si può comprimere in questo modo.
Se nella view non istanzio il form che devo usare, ma le dico in quale classe è definito, io non devo fare più niente.

In `polls/urls.py`:

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

Il `template`

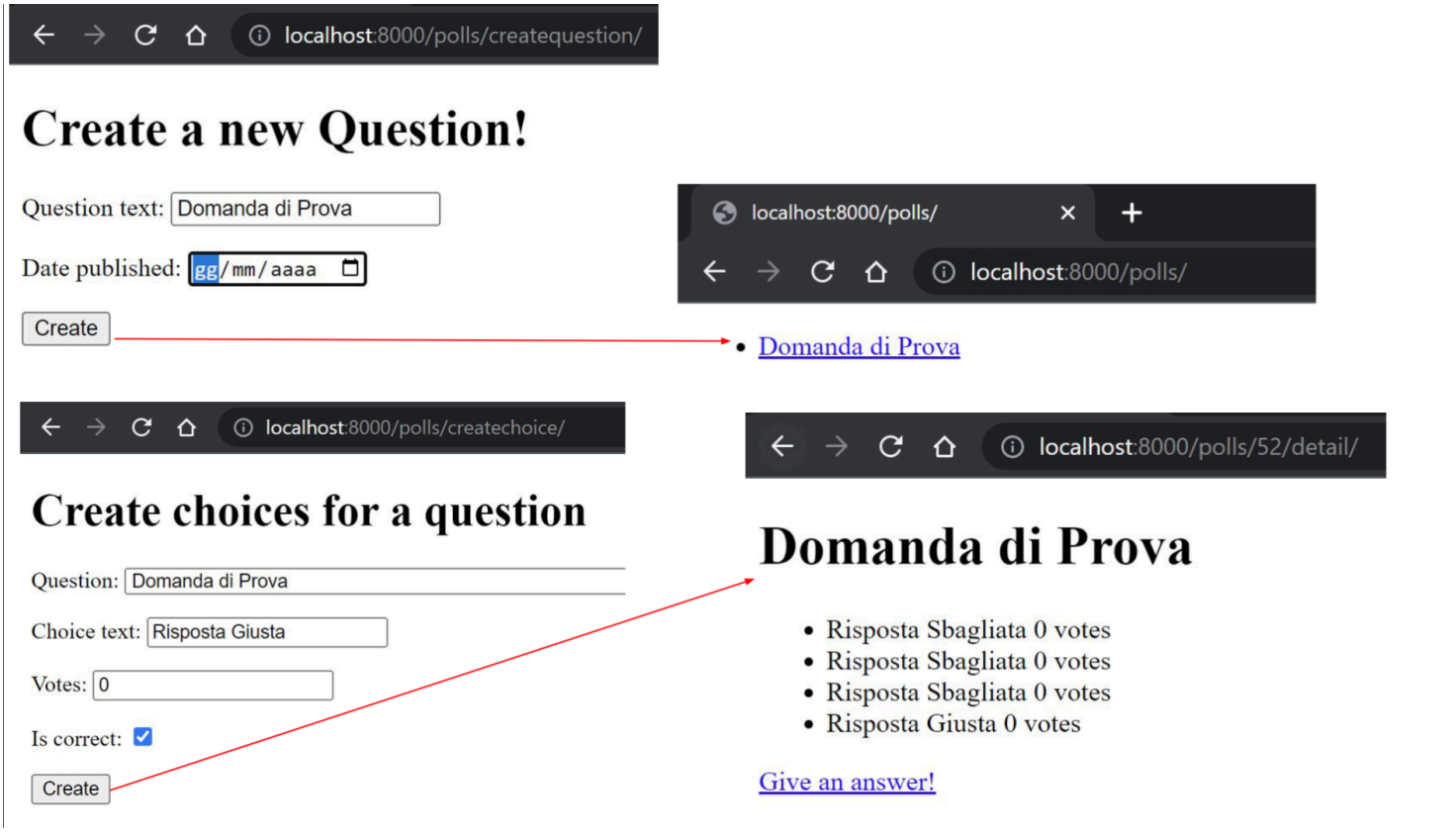
In `template/polls/createentry.html`:

```
{% 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 %}
```

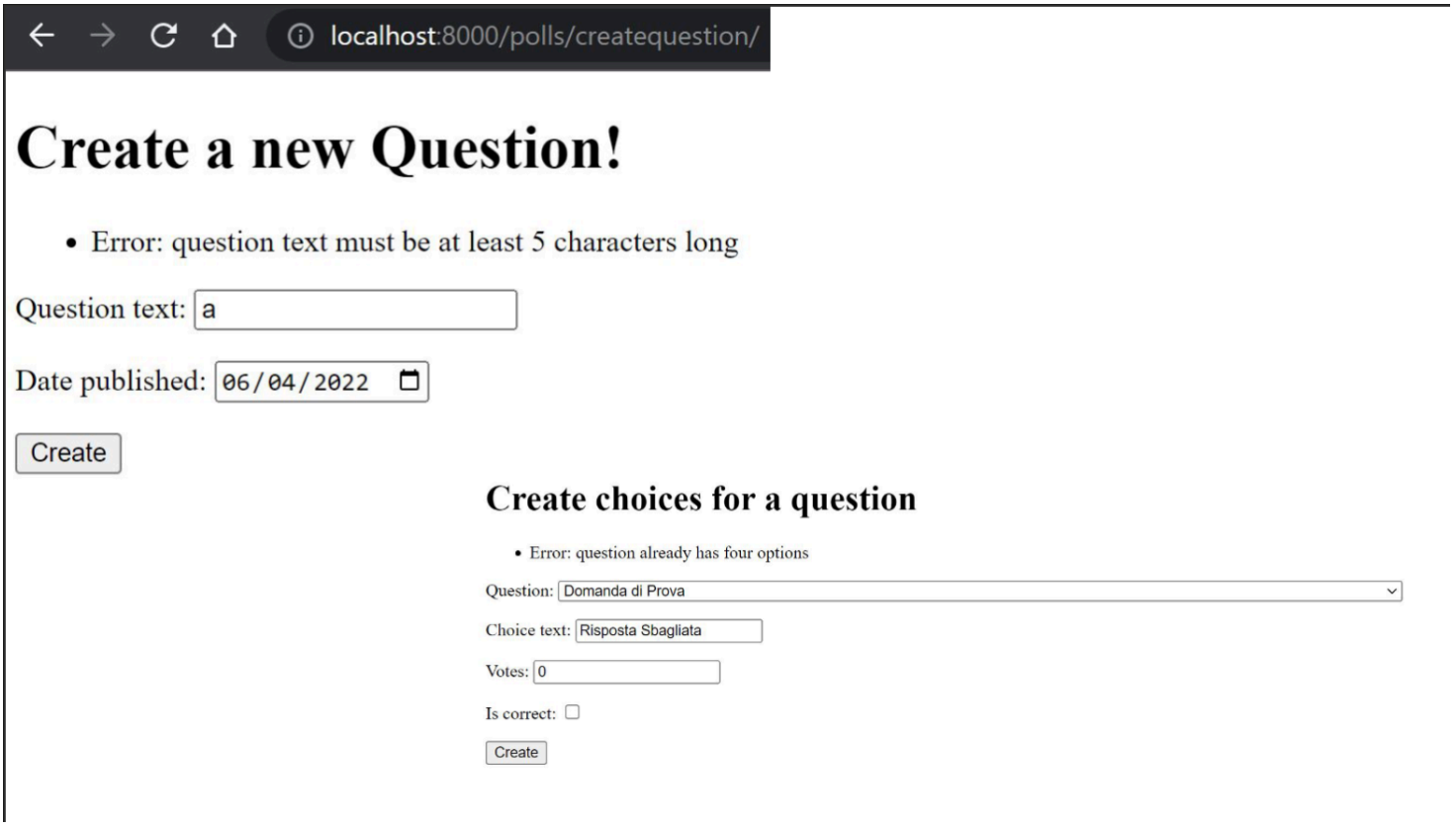
Nota che nel blocco titolo abbiamo passato come variabile di contesto la variabile descrizione definita nelle classi dei ModelForms.

I risultati

Right question and choices



Wrong question and choices



Django Crispy Forms (le slide sono fatte più che bene)