

Django CBV

Class Based Views

Riassunto

- Abbiamo visto come sfruttare l'intero design pattern di Django
 - Models (Database relazionali tramite ORM + migrations)
 - views, intese come funzioni che regolano la logica della nostra webapp
 - e Templates, per il lato presentazione

Problemi?

Da un punto di vista funzionale, no, nessun problema.

Da un punto di vista di comodità di sviluppo invece ci siamo accorti di come le nostre functions views siano sostanzialmente un ammasso di copia & incolla, con parecchio codice ripetuto con poche variazioni tra una function views e l'altra.

Un sacco di codice rimane necessario per tutte le operazioni di accesso dei DB.

Ma vi avevo promesso che Django sostiene la policy DRY, do not repeat yourself!

DRY in Django

Django rimane fedele alla policy DRY, ma noi sviluppatori dobbiamo capire come sfruttare questa sua proprietà.

Rispondere ad una richiesta del client con una function view, per esempio, non sempre ci consente di sfruttare la policy DRY.

Ma django è scritto in python, il quale è un linguaggio di programmazione orientato agli oggetti.

Le **classi** di solito costituiscono un buon approccio alle metodologie di sviluppo basate sul riutilizzo di codice già scritto.

E se quindi rispondessimo ad una richiesta tramite una classe anziché una funzione?

CBV vs FBV (Class Based Views vs. Function Based Views)

	FBV	CBV
Cosa usa?	Funzioni	Metodi\Classi\Oggetti
Riutilizzabilità	Bassa: molto codice ripetuto tra le varie function views	Alta: poco o nessun codice ripetuto tra View imparentate tra di loro
Decorators	Supportati	Si e no. Si passa per i <i>mixin</i> , quindi Multiple Inheritance
Accesso ai DB: operazioni di base	Molto codice esplicito, difficilmente riutilizzabile	Possibilità di ereditare da classi presenti nei moduli\pkg di django: sostanziale riduzione del codice
Generiche operazioni personalizzate	Probabilmente la scelta migliore quando le logiche da implementare aumentano nella loro complessità	Partendo dalle View “date in regalo” da django, alcune particolari operazioni diventano difficilmente implementabili

Morale della favola

Vedremo situazioni in cui usare una class based view è senza dubbio la cosa migliore da fare, ma esistono situazioni e casi particolari in cui le function views paradossalmente ci possono risparmiare un sacco di mal di testa.

In altre parole, non pensate a CBV e FBV come scelte “radicali”, in quanto sono meccanismi complementari e non esclusivi.

Applicazione di esempio

La trovate nel git in

...django/CBVprj/...

Un semplice gestore di **Studenti** ed **Insegnamenti**. Studenti ed insegnamenti possono essere aggiunti/modificati/rimossi etc...

Gli studenti possono iscriversi a diversi insegnamenti...

La solita premessa

```
mkdir CBVprj
```

```
cd CBVprj
```

```
pipenv install django
```

```
pipenv shell
```

```
django-admin startproject cbvprj
```

```
cd cbvprj
```

```
python manage.py runserver
```

si butta giù il webserver e poi..

```
python manage.py migrate
```


Creiamo un'app

Creiamo un app nel nostro progetto.

La chiamiamo **iscrizioni**.

```
python manage.py startapp iscrizioni
```

Occorre ora sistemare un paio di cose in settings.py.

In particolare, aggiungere iscrizioni in INSTALLED_APPS e aggiungere:

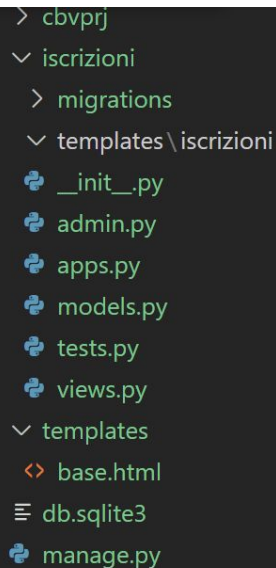
```
os.path.join(BASE_DIR, "templates")
```

nella chiave DIRS di TEMPLATES

Altre operazioni iniziali

aggiungere le cartelle templates, sia nel root project, che come sottocartella della nostra app. Ricordiamoci del trucco per evitare conflitti di naming.

Copiamo ed incolliamo il nostro base.html nel templates della root.



```
> cbvprj
  ▾ iscrizioni
    > migrations
    ▾ templates \ iscrizioni
      + __init__.py
      + admin.py
      + apps.py
      + models.py
      + tests.py
      + views.py
    ▾ templates
      <> base.html
      ≡ db.sqlite3
      + manage.py
```

Creiamo i modelli (iscrizioni/models.py)

```
class Studente(models.Model):
    name = models.CharField(max_length=50)
    surname = models.CharField(max_length=50)

    def __str__(self):
        return "ID: " + str(self.pk) + ": " + self.name + " " + self.surname

    class Meta:
        verbose_name_plural = "Studenti"

class Insegnamento(models.Model):
    titolo = models.CharField(max_length=50)
    studenti = models.ManyToManyField(Studente,default=None)

    def __str__(self):
        return "ID: " + str(self.pk) + ": " + self.titolo

    class Meta:
        verbose_name_plural = "Insegnamenti"
```

init_db & erase_db (initcmds.py, funzioni chiamate in urls.py)

```
cbvprj > initcmds.py > init_db
1 from iscrizioni.models import Studente, Insegnamento
2
3 def erase_db():
4     print("Cancello il DB")
5     Studente.objects.all().delete()
6     Insegnamento.objects.all().delete()
```

```
def init_db():
    if Studente.objects.all().count() > 0:
        return

    lista_studenti = [
        ("Mario", "Bianchi"),
        ("Luigi", "Verdi"),
        ("Giovanni", "Rossi"),
        ("Maria", "Viola"),
        ("Antonia", "Azzurri")
    ]

    lista_insegnamenti = [
        "Programmazione ad Oggetti",
        "Programmazione Mobile",
        "Tecnologie Web",
        "Cloud and Edge Computing",
        "Internet, Web e Cloud"
    ]
```

```
for s in lista_studenti:
    s_db = Studente()
    s_db.name = s[0]
    s_db.surname = s[1]
    s_db.save()

studenti = Studente.objects.all()

for index, ins in enumerate(lista_insegnamenti):
    ins_db = Insegnamento()
    ins_db.titolo = ins
    ins_db.save()

    if index == 0: continue
    count = 0
    for s in studenti:
        ins_db.studenti.add(s)
        count += 1
        if count > index:
            break

print("DUMP DB")
print("Studenti")
for s in studenti:
    print(s)
print("Insegnamenti")
for i in Insegnamento.objects.all():
    print(i)
    print("Studenti iscritti " + str(i.studenti.all()))
```

DUMP DB

Studenti

ID: 66: Mario Bianchi

ID: 67: Luigi Verdi

ID: 68: Giovanni Rossi

ID: 69: Maria Viola

ID: 70: Antonia Azzurri

Insegnamenti

ID: 38: Programmazione ad Oggetti

Studenti iscritti <QuerySet []>

ID: 39: Programmazione Mobile

Studenti iscritti <QuerySet [<Studente: ID: 66: Mario Bianchi>, <Studente: ID: 67: Luigi Verdi>]>

ID: 40: Tecnologie Web

Studenti iscritti <QuerySet [<Studente: ID: 66: Mario Bianchi>, <Studente: ID: 67: Luigi Verdi>, <Studente: ID: 68: Giovanni Rossi>]>

ID: 41: Cloud and Edge Computing

Studenti iscritti <QuerySet [<Studente: ID: 66: Mario Bianchi>, <Studente: ID: 67: Luigi Verdi>, <Studente: ID: 68: Giovanni Rossi>, <Studente: ID: 69: Maria Viola>]>

ID: 42: Internet, Web e Cloud

Studenti iscritti <QuerySet [<Studente: ID: 66: Mario Bianchi>, <Studente: ID: 67: Luigi Verdi>, <Studente: ID: 68: Giovanni Rossi>, <Studente: ID: 69: Maria Viola>, <Studente: ID: 70: Antonia Azzurri>]>

Ricordiamoci anche...

Di creare un superutente admin

python manage.py createsuperuser

ed in iscrizione/admin.py

```
from .models import Studente, Insegnamento  
admin.site.register(Studente)  
admin.site.register(Insegnamento)
```

Possiamo cominciare: Metodi CRUD sui DB

I metodi CRUD rappresentano le operazioni base su cui operare nei nostri database.

In particolare vogliamo essere in grado di Creare, Leggere, Aggiornare ed eventualmente eliminare entry delle nostre tabelle.

Lo abbiamo visto con FBV, ora vedremo come ottenere lo stesso risultato con le django CBV.

Django e le sue generic Views

Django ci mette a disposizione delle “**View**” pre-confezionate per i metodi CRUD.

Tali View sono quindi classi. Appartengono ai moduli *django.views.generic.**

- Si crea quindi una **classe** che **estende** dalle view “generic” di Django
- In particolare, **vi è una View per ogni operazione CRUD**.
 - **ListView**: elenca le entry delle nostre tabelle
 - **UpdateView**: ci permette di modificare una specifica entry di una tabella
 - **CreateView**: ci permette di creare una entry di una tabella
 - **DeleteView**: ci permette di rimuovere una entry di una tabella
 - **DetailView**: ci permette di leggere gli attributi di una specifica entry di una tabella.
- Nella nostra classe si specifica a quale **Model** ci si riferisce
- Si specifica il **template** per renderizzare in HTML/DTL
- ...e abbiamo già finito...

Proviamo

In iscrizione/views.py

```
from django.views.generic.list import ListView  
from .models import *  
  
# Create your views here.  
  
class ListaStudentiView(ListView):  
    model = Studente  
    template_name = "iscrizioni/lista_studenti.html"
```

Cosa manca?

Il template. Lo faremo tra un attimo.

Inoltre, dobbiamo rendere accessibile la nostra View.

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

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

#erase_db()
init_db()
```

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

app_name = "iscrizioni"

urlpatterns = [
    path("listastudenti/", views.ListaStudentiView.as_view(), name="listastudenti"),
]
```

root: urls.py

iscrizioni/urls.py (occorre crearlo)

as_view()

Attenzione.

Quando si elencano gli url-patterns, c'è questa piccola differenza tra function views e class based views. La funzione *path* si aspetta di lavorare con delle funzioni.

as_view è un metodo della classe **View** di django. **Tutte** le View dei moduli di Django ereditano da questa classe. Restituisce un **oggetto funzione** che incorpora i meccanismi della view...

Il template: lista_studenti.html

in iscrizioni/template/iscrizioni

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

{% block title %} Dump del DB Studenti {% endblock %}

{% block content %}

{% for p in object_list %}
<ul>
    {{ p }}
</ul>
{% endfor %}

{% endblock %}
```

Il template: lista_studenti.html

in iscrizioni/template/iscrizioni

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

```
{% block title %} Dump del DB Studenti {% endblock %}
```

```
{% block content %}
```

```
{% for p in object_list %}
```


```
<ul>
```

```
    {{ p }}
```

```
</ul>
```

```
{% endfor %}
```

```
{% endblock %}
```



Variabile di contesto ottenuta “**gratis**”. Si chiama **object_list** e contiene il **QuerySet** operato sul modello specificato nella nostra estensione della **ListView**

Su localhost:8000/iscrizioni/listastudenti/

← → ↻ 🏠 ⓘ localhost:8000/iscrizioni/listastudenti/

ID: 66: Mario Bianchi

ID: 67: Luigi Verdi

ID: 68: Giovanni Rossi

ID: 69: Maria Viola

ID: 70: Antonia Azzurri

Se ne deduce che `{{ p }}` inteso come elemento di `object_list` viene renderato chiamando il metodo magico **str**, così come definito nell'implementazione nel nostro **Studente Model**

Riassumendo

- La nostra View ha 2/3 righe di codice. Decisamente meno verbose di quanto avremmo dovuto fare con una function view equivalente:

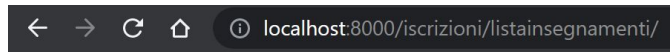
#function view equivalente alla CBV ListaStudentiView:

```
def lista_studenti_function(request):  
    lista = Studente.objects.all()  
    templ = "iscrizioni/lista_studenti.html"  
    ctx = {"object_list":lista}  
    return render(request,template_name=templ,context=ctx)
```

Esercizio 1

Creare una derivazione di ListView che elenca gli insegnamenti.

Si cerchi di ottenere il seguente output:



Programmazione ad Oggetti
Nessuno studente iscritto

Programmazione Mobile
1. Mario Bianchi
2. Luigi Verdi

Tecnologie Web
1. Mario Bianchi
2. Luigi Verdi
3. Giovanni Rossi

Cloud and Edge Computing
1. Mario Bianchi
2. Luigi Verdi
3. Giovanni Rossi
4. Maria Viola

Internet, Web e Cloud
1. Mario Bianchi
2. Luigi Verdi
3. Giovanni Rossi
4. Maria Viola
5. Antonia Azzurri

...ma una CBV ha anche lati negativi...

Il fatto di aver semplicemente indicato il model tramite attributo di classe implica che di default non possiamo operare un pre-processing della nostra lista di studenti.

Cosa che sarebbe stata banale, e lo abbiamo già visto, tramite le FBV.

Esempio: voglio elencare gli insegnamenti in cui **vi è almeno uno studente iscritto**.

In realtà c'è modo di operare dei filtri anche tramite le CBV...

Alcuni metodi fondamentali delle CBV

Sono ereditati da *View*, il capostipite di tutte le View preconfezionate di django.

- `get_queryset`
 - Permette di fare operazioni sul queryset sulla tabella indicata dall'attributo `model`
- `get_context_data`
 - Permette di aggiungere variabili di contesto

Elenca studenti iscritti ad almeno un corso (CBV)

```
class ListaInsegnamentiAttivi(ListView):  
    model = Insegnamento  
    template_name = "iscrizioni/insegnamenti_attivi.html"  
  
    def get_queryset(self):  
        return self.model.objects.exclude(studenti__isnull=True)  
  
    def get_context_data(self, **kwargs):  
        context = super().get_context_data(**kwargs)  
        context['titolo'] = "Insegnamenti Attivi"  
        return context
```

iscrizioni/insegnamenti_attivi.html

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

{% block title %} {{titolo}} {% endblock %}

{% block content %}

<h1> {{titolo}} </h1>

{% for p in object_list %}
<ul>
    {{ p }}
</ul>
{% endfor %}

{% endblock %}
```

iscrizioni/insegnamenti_attivi.html

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

```
{% block title %} {{titolo}} {% endblock %}
```

```
{% block content %}
```

```
<h1> {{titolo}} </h1>
```

```
{% for p in object_list %}
```

```
<ul>
```

```
    {{ p }}
```

```
</ul>
```

```
{% endfor %}
```

```
{% endblock %}
```

Variabile di contesto aggiunta

Variabile di contesto ereditata
da ListView

Programmazione ad Oggetti

Nessuno studente iscritto

Programmazione Mobile

1. Mario Bianchi
2. Luigi Verdi

Tecnologie Web

1. Mario Bianchi
2. Luigi Verdi
3. Giovanni Rossi

Cloud and Edge Computing

1. Mario Bianchi
2. Luigi Verdi
3. Giovanni Rossi
4. Maria Viola

Internet, Web e Cloud

1. Mario Bianchi
2. Luigi Verdi
3. Giovanni Rossi
4. Maria Viola
5. Antonia Azzurri

Insegnamenti Attivi

ID: 39: Programmazione Mobile

ID: 40: Tecnologie Web

ID: 41: Cloud and Edge Computing

ID: 42: Internet, Web e Cloud

Contesto ereditato

Cos'altro possiamo trovare nella variabile context ereditata da ListView?

```
def get_context_data(self, **kwargs):  
    context = super().get_context_data(**kwargs)  
    print(context.keys())
```

OUTPUT:

```
dict_keys(['paginator', 'page_obj', 'is_paginated',  
'object_list', 'insegnamento_list', 'view'])
```

La variabile view

La variabile view ci permette di accedere a metodi e attributi del nostro derivato di ListView.

Questo ci consente di definire ulteriori logiche arbitrarie e di chiamarle lato template.

Esempio: si faccia una CBV che elenca gli studenti e restituisca il numero totale di iscrizioni.


```

class ListaStudentiIscritti(ListView):
    model = Studente
    template_name = "iscrizioni/studenti_iscritti.html"

    def get_model_name(self):
        return self.model._meta.verbose_name_plural

    def get_context_data(self, **kwargs):
        ctx = super().get_context_data(**kwargs)
        ctx["titolo"] = "Lista Studenti con Iscrizione"
        return ctx

    def get_totale_iscrizioni(self):
        count = 0
        for i in Insegnamento.objects.all():
            count += i.studenti.all().count()
        return count

```

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

```
{% block title %} {{titolo}} {% endblock %}
```

```
{% block content %}
```

```
<h1> Lista di {{ view.get_model_name }} </h1>
```

```
{% for p in object_list %}
```

```
<ul>
```

```
    {{ p.name }} {{ p.surname }} |
```

```
</ul>
```

```
{% endfor %}
```

```
<p> Il numero totale di iscrizioni è {{ view.get_totale_iscrizioni }} </p>
```

```
{% endblock %}
```

```
class ListaStudentiIscritti(ListView):  
    model = Studente  
    template_name = "iscrizioni/studenti_iscritti.html"
```

```
def get_model_name(self):  
    return self.model._meta.verbose_name_plural
```

```
def get_context_data(self, **kwargs):  
    ctx = super().get_context_data(**kwargs)  
    ctx["titolo"] = "Lista Studenti con Iscrizione"  
    return ctx
```

```
def get_totale_iscrizioni(self):  
    count = 0  
    for i in Insegnamento.objects.all():  
        count += i.studenti.all().count()  
    return count
```

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

```
{% block title %} {{titolo}} {% endblock %}
```

```
{% block content %}
```

```
<h1> Lista di {{ view.get_model_name }} </h1>
```

```
{% for p in object_list %}
```

```
<ul>
```

```
    {{ p.name }} {{ p.surname }} |
```

```
</ul>
```

```
{% endfor %}
```

```
<p> Il numero totale di iscrizioni è {{ view.get_totale_iscrizioni }} </p>
```

```
{% endblock %}
```

Lista di Studenti

Mario Bianchi

Luigi Verdi

Giovanni Rossi

Maria Viola

Antonia Azzurri

Il numero totale di iscrizioni è 14

Programmazione ad Oggetti
Nessuno studente iscritto

Programmazione Mobile
1. Mario Bianchi
2. Luigi Verdi

Tecnologie Web
1. Mario Bianchi
2. Luigi Verdi
3. Giovanni Rossi

Cloud and Edge Computing
1. Mario Bianchi
2. Luigi Verdi
3. Giovanni Rossi
4. Maria Viola

Internet, Web e Cloud
1. Mario Bianchi
2. Luigi Verdi
3. Giovanni Rossi
4. Maria Viola
5. Antonia Azzurri

Riassumendo

Tramite le CBV si aprono molte possibilità per personalizzare ulteriormente la logica della nostra web app.

Tramite la variabile “view” che appare nel contesto del template, possiamo quindi andare oltre i metodi già presenti nelle View di Django.

Però occorre fare attenzione: le possibilità non sono propriamente infinite. Per esempio non è possibile chiamare dei view methods con parametri da template.

Questo per una scelta filosofica degli sviluppatori di django.

Dagli sviluppatori di django...

#16146 closed New feature (wontfix)

Calling functions with arguments inside a template

Resolution: → **wontfix**

Status: new → **closed**

Django actually prevents this by design. The idea is to keep most of the logic in views and keep the template language as simple as possible (so that it can easily be used by non tech-savvy designers, for example). Quoting from the doc:

"Because Django intentionally limits the amount of logic processing available in the template language, it is not possible to pass arguments to method calls accessed from within templates. Data should be calculated in views, then passed to templates for display." (⇒ <https://docs.djangoproject.com/en/1.3/topics/templates/#accessing-method-calls>)

So you'll have to do that in the view. Writing a custom template tag or filter might also be of good help. This is a common problem, so don't hesitate to ask on the django-users mailing list for more advice.

CreateView

Siamo quindi in grado di interrogare le tabelle di un DB.

Vediamo quindi ora come creare nuove entry alle tabelle.

Si parte ancora una volta, da una View ereditata dal solito package django.

`django.views.generic`, in particolare si importerà

`django.views.generic.edit.CreateView`

Cosa si specifica in una CreateView?

Ovviamente il model ed il template.

Si aggiunge l'attributo *fields* per indicare quali attributi si voglia permettere al client di impostare.

Si aggiunge l'attributo *success_url* per indicare l'url di redirectionamento in caso di scrittura sul DB avvenuta a buon fine.

In iscrizioni/views.py

```
class CreateStudenteView(CreateView):  
    model = Studente  
    template_name = "iscrizioni/crea_studente.html"  
    fields = "__all__"  
    success_url = reverse_lazy("iscrizioni:listastudenti")
```

In iscrizioni/urls.py

```
app_name = "iscrizioni"  
  
urlpatterns = [  
    # ...  
    path("creastudente/", views.CreateStudenteView.as_view(), name="creastudente")  
]
```


reverse & reverse_lazy: (from django.urls)

Eseguono un “reverse look up”. Ossia, anzichè specificare un url “hardcodato”, abbiamo qui la possibilità di usare l’alias del nostro url pattern. Tale alias è tipicamente composto così: “*app_name:url_name*”. Da qui, il motore di django si assicurerà di creare un pattern in grado di fare match con l’url di redirectione voluto.

E’ necessario fare questo reverse lookup in quanto più string patterns possono fare match con lo stesso url. Inoltre, evitando di esplicitare direttamente l’url senza passare per il suo name alias mi evita di dovermi ricordare che se cambio l’url in urls.py allora devo anche cambiare il valore di success_url.

reverse & reverse_lazy: altri dettagli

reverse si usa nei metodi e nelle funzioni. Restituisce una stringa

reverse_lazy si usa invece al momento di assegnare valori ad una variabile o attributo. Restituisce un oggetto complesso.

Come suggerisce il nome, “lazy” indica che questo reverse lookup verrà eseguito tardivamente. Confondere **reverse** e **reverse_lazy** può causare degli errori (**reverse not found**, o altre tipologie di errori collegati).

Siamo costretti a capire questo meccanismo in quanto è legato a come funziona la direttiva import di python e a quando vengono inizializzati gli attributi di una classe. In particolare, python inizializza gli attributi della classe **prima di importare e risolvere gli url patterns**. Quindi, se non ritardassi il reverse lookup, il name alias di success_url **rischia di non trovare nessuna corrispondenza**

Esempio utilizzo *reverse*

La seguente View è equivalente alla CreateStudenteView vista prima.

```
class CreateStudenteView(CreateView):  
    model = Studente  
    template_name = "iscrizioni/crea_studente.html"  
    fields = "__all__"  
    #success_url = reverse_lazy("iscrizioni:listastudenti")  
  
    #metodo ereditato dalla classe padre  
    def get_success_url(self):  
        return reverse("iscrizioni:listastudenti")
```

Il template

(iscrizioni/templates/iscrizioni/crea_studente.html)

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

{% block title %} Crea Studente {% endblock %}

{% block content %}

<h1> Crea Studente </h1>

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

    {{ form.as_p }}

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

</form>

{% endblock %}
```

Cerchiamo di capire

I campi della tabella che sto cercando di modificare sono inclusi da una context variabile chiamata *form*. La quale deve essere compresa all'interno dei tag HTML `<form>... </form>`.

Contrariamente ai form che abbiamo completamente specificato “manualmente” in precedenza, qui si specifica che la sottomissione avviene tramite richiesta **POST**, e non GET.

La variabile *form* esplicita quindi i *fields* da noi definiti nella View in python, e li renderizza come se fossero `<p></p>`, ossia paragraph in HTML. Questo perchè abbiamo scritto ***form.as_p*** all'interno dei tag `<form>`

form as

*{{ **form.as_p** }}* Rendera sottoforma di paragrafi, quindi compresa tra i tag <p>

*{{ **form.as_table** }}* Rendera sottoforma di tabella, quindi compresa tra i tag <tr>

*{{ **form.as_ul** }}* Rendera tramite una item list, quindi compresa nei tag

GET vs POST (da <https://www.w3schools.com/>)

	GET	POST
BACK button/Reload	Harmless	Data will be re-submitted (the browser should alert the user that the data are about to be re-submitted)
Bookmarked	Can be bookmarked	Cannot be bookmarked
Cached	Can be cached	Not cached
Encoding type	application/x-www-form-urlencoded	application/x-www-form-urlencoded or multipart/form-data. Use multipart encoding for binary data
History	Parameters remain in browser history	Parameters are not saved in browser history
Restrictions on data length	Yes, when sending data, the GET method adds the data to the URL; and the length of a URL is limited (maximum URL length is 2048 characters)	No restrictions
Restrictions on data type	Only ASCII characters allowed	No restrictions. Binary data is also allowed
Security	GET is less secure compared to POST because data sent is part of the URL Never use GET when sending passwords or other sensitive information!	POST is a little safer than GET because the parameters are not stored in browser history or in web server logs
Visibility	Data is visible to everyone in the URL	Data is not displayed in the URL

In django...

Similmente a quanto visto con GET,

request.GET["param_name"]

ora sarà

request.POST["param_name"]


```
{% csrf_token %}
```

E' un meccanismo di sicurezza che siamo obbligati a specificare nel DTL.

E' reso possibile perché in settings.py, di default è listato il
django.middleware.csrf.CsrfViewMiddleware nella variabile MIDDLEWARE.

Evita il **CROSS SITE REFERENCE FORGERY**, uno scenario di attacco
dettagliato nella prossima slide.

Cross Site Reference Forgery

Se non ci si proteggesse contro il csrf, una potenziale vittima esegue il login nel nostro sito. Il cookie di autenticazione è salvato nel nostro browser. Una volta fatto ciò, un utente malevolo può indurre la vittima a cliccare su un link verso un url del nostro sito, come quello per modificare i DB. L'utente malevolo può cambiare i parametri della richiesta della vittima, quindi operando modifiche non volute e dannose. Il **csrf token previene tutto ciò associando la prima parte della richiesta ad un token autogenerato, mandando tale stringa al client ma tenendosi il token lato server**. Se un client compromesso provasse a modificare il DB non mandando o mandando un token non presente nel server, quest'ultimo blocca le operazioni.

localhost:8000/iscrizioni/creastudente/

Crea Studente

Name:

Surname:

Save

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body> == $0
    <h1> Crea Studente </h1>
    <form method="post">
      <input type="hidden" name="csrfmiddlewaretoken" value="MCoiTzaC1Nrhq7kcQFEBAUB1
      0URv9gKI0jkh0oPbjNJZJIitTZDuTE0qV15LpLja">
      <p>
        <label for="id_name">Name:</label>
        <input type="text" name="name" maxlength="50" required id="id_name">
      </p>
      <p>
        <label for="id_surname">Surname:</label>
        <input type="text" name="surname" maxlength="50" required id="id_surname">
      </p>
      <input type="submit" value="Save">
    </form>
  </body>
</html>
```

Visuale “Strumenti per Sviluppatori” del browser

Giochiamo a fare i piccoli hacker

Apriamo la console degli sviluppatori del nostro browser di fiducia:

- Proviamo a cambiare il csrf token
- Bypassiamo il limite di caratteri per i campi dello studente

Togliere o modificare caratteri nel token

Forbidden (403)

CSRF verification failed. Request aborted.

Help

Reason given for failure:

CSRF token from POST has incorrect length.

Help

Reason given for failure:

CSRF token from POST incorrect.

In general, this can occur when there is a genuine Cross Site Request Forgery, or when [Django's CSRF mechanism](#) has not been used correctly. For POST forms, you need to ensure:

- Your browser is accepting cookies.
- The view function passes a request to the template's [render](#) method.
- In the template, there is a `{% csrf_token %}` template tag inside each POST form that targets an internal URL.
- If you are not using `CsrfViewMiddleware`, then you must use `csrf_protect` on any views that use the `csrf_token` template tag, as well as those that accept the POST data.
- The form has a valid CSRF token. After logging in in another browser tab or hitting the back button after a login, you may need to reload the page with the form, because the token is rotated after a login.

You're seeing the help section of this page because you have `DEBUG = True` in your Django settings file. Change that to `False`, and only the initial error message will be displayed.

You can customize this page using the `CSRF_FAILURE_VIEW` setting.

Bypass lunghezza stringhe

E' il campo "max_length" delle nostre input string nel form. Il suo valore è basato sui parametri immessi in fase di costruzione della classe che eredita da Model (in questo caso Studente, ed era 50)

Crea Studente

- Ensure this value has at most 50 characters (it has 70).

Name:

Surname:

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <h1> Crea Studente </h1>
    <form method="post">
      <input type="hidden" name="csrfmiddlewaretoken"
        5xuTuFepU7ssK2MufBusda1GbJZ8PrT00YI9KaNR">
      <p>
        <label for="id_name">Name:</label>
        ... <input type="text" name="name" maxlength="70"
          </p>
        <p>...</p>
        <input type="submit" value="Save">
      </form>
    </body>
  </html>
```

Altri controlli

Crea Studente

- Ensure this value has at most 50 characters (it has 70).

Name:

Surname:



Compila questo campo.

Save

Controllo aggiuntivo lato server

Controllo lato client/browser

success_url

← → ↻ ⓘ localhost:8000/iscrizioni/creastudente/

Crea Studente

Name:

Surname:

Save

← → ↻ ⓘ localhost:8000/iscrizioni/listastudenti/

ID: 66: Mario Bianchi

ID: 67: Luigi Verdi

ID: 68: Giovanni Rossi

ID: 69: Maria Viola

ID: 70: Antonia Azzurri

ID: 76: Nuovo Studente

Campi “complessi”

Come esercizio si provi a fare l'equivalente di `CreateStudentView` di `CreateInsegnamentoView`.

Si noti come `Insegnamento` abbia il campo `studenti`, il quale è stata modellata come una relazione `ManyToMany`.

L'url di successo dell'operazione deve portare alla lista degli insegnamenti.

Risultato Atteso

← → ↻ ⓘ localhost:8000/iscrizioni/creainsegnamento/

Crea Insegnamento

Titolo: Advanced GPU Programmii

Studenti:

ID: 66: Mario Bianchi

ID: 67: Luigi Verdi

ID: 68: Giovanni Rossi

ID: 69: Maria Viola

Save

← → ↻ ⓘ localhost:8000/iscrizioni/listainsegnamenti/

Programmazione ad Oggetti
Nessuno studente iscritto

Programmazione Mobile
1. Mario Bianchi
2. Luigi Verdi

Tecnologie Web
1. Mario Bianchi
2. Luigi Verdi
3. Giovanni Rossi

Cloud and Edge Computing
1. Mario Bianchi
2. Luigi Verdi
3. Giovanni Rossi
4. Maria Viola

Internet, Web e Cloud
1. Mario Bianchi
2. Luigi Verdi
3. Giovanni Rossi
4. Maria Viola
5. Antonia Azzurri

Advanced GPU Programming
1. Mario Bianchi
2. Giovanni Rossi
3. Maria Viola

Il codice

Zero differenze sostanziali. Sia in iscrizioni/views.py che nel DTL associato

```
class CreateInsegnamentoView(CreateView):  
    model = Insegnamento  
    template_name = "iscrizioni/crea_insegnamento.html"  
    fields = "__all__"  
    success_url = reverse_lazy("iscrizioni:listainsegnamenti")
```

Il codice

Zero differenze sostanziali. Sia in iscrizioni/views.py che nel DTL associato

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

{% block title %}Crea Insegnamento{% endblock %}

{% block content %}

<h1> Crea Insegnamento </h1>

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

    {{ form.as_p }}

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

</form>

{% endblock %}
```

DetailView

Ci permette di fare una query ad una tabella partendo dalla sua primary key.

Ciò ci permette di avere un contesto in cui esiste la variabile “object”, attraverso la quale possiamo poi scegliere quali campi visualizzare e come all’interno del template.

La DetailView è una classe base di django, importabile da *django.views.generic.detail*

DetailInsegnamentoView in iscrizione/views.py

```
class DetailInsegnamentoView(DetailView):  
    model = Insegnamento  
    template_name = "iscrizioni/insegnamento.html"
```

Non occorre specificare altro: dalla chiave si ottiene un oggetto. Nel template sceglieremo noi i fields da vedere.

La chiave

Si specifica nell'url. In iscrizioni/urls.py:

```
path("insegnamento/<pk>/",  
      views.DetailInsegnamentoView.as_view(),  
      name="insegnamento")
```

Template

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

{% block title %} {{object.titolo}} {% endblock %}

{% block content %}

<h1>Dettagli di {{object.titolo}}</h1>

<br>

    {% if object.studenti.all.count == 0 %}

        Nessuno studente iscritto

    {% else %}

        {% for s in object.studenti.all %}

            <li> {{s.name}} {{s.surname}} </li>

        {% endfor %}

    {% endif %}

{% endblock %}
```


← → ↻ ⓘ localhost:8000/iscrizioni/insegnamento/41/

Dettagli di Cloud and Edge Computing

- Mario Bianchi
- Luigi Verdi
- Giovanni Rossi
- Maria Viola

UpdateView (django.views.generic.edit)

Accessibilità simile a DetailView:

Stesso metodo di passaggio della chiave per ottenere una entry di una tabella (tramite url)

Modifica simile a CreateView:

Stesso metodo per restituire un form lato template e sottomissione valori modificati tramite richiesta POST, protetta con csrf token.

Attributi richiesti: `template_name`, `model`, `success_url` & `fields`.

In iscrizioni/views.py UpdateInsegnamentoView

```
class UpdateInsegnamentoView(UpdateView):  
    model = Insegnamento  
    template_name = "iscrizioni/edit_insegnamento.html"  
    fields = "__all__"  
  
    def get_success_url(self):  
        pk = self.get_context_data()["object"].pk  
        return reverse("iscrizioni:insegnamento",kwargs={'pk': pk})
```

Modifica Insegnamento Programmazione ad Oggetti

Titolo:

Studenti:

ID: 68: Giovanni Rossi

ID: 69: Maria Viola

ID: 70: Antonia Azzurri

ID: 76: Nuovo Studente

Dettagli di Programmazione ad Oggetti

- Nuovo Studente

DeleteView (django.views.generic.edit)

Come Create ed Update view per ottenere la primary key (pk) dell'elemento da cancellare.

Occorre avere comunque un form di conferma della cancellazione all'interno del template associato alla View.

```

class DeleteEntitaView(DeleteView):

    template_name = "iscrizioni/cancella_entry.html"

    def get_context_data(self, **kwargs):
        ctx = super().get_context_data()
        entita = "Studente"
        if self.model == Insegnamento:
            entita = "Insegnamento"
        ctx["entita"] = entita
        return ctx

    def get_success_url(self):
        if self.model == Studente:
            return reverse("iscrizioni:listastudenti")
        else:
            return reverse("iscrizioni:listainsegnamenti")

class DeleteStudenteView(DeleteEntitaView):
    model = Studente

class DeleteInsegnamentoView(DeleteEntitaView):
    model = Insegnamento

```

Essendo le due Views molto simili, possiamo sfruttare l'ereditarietà per risparmiarci di (ri)-scrivere qualche linea di codice...

← → ↻ ⓘ localhost:8000/iscrizioni/cancellastudente/76/

Cancellazione di Studente

Sicuro di voler cancellare "ID: 76: Nuovo Studente"?

Conferma

← → ↻ ⓘ localhost:8000/iscrizioni/listastudenti/

ID: 66: Mario Bianchi

ID: 67: Luigi Verdi

ID: 68: Giovanni Rossi

ID: 69: Maria Viola

ID: 70: Antonia Azzurri



localhost:8000/iscrizioni/cancellainsegnamento/43/

Cancellazione di Insegnamento

Sicuro di voler cancellare "ID: 43: Advanced GPU Programming"?

Conferma



localhost:8000/iscrizioni/listainsegnamenti/

Programmazione ad Oggetti
Nessuno studente iscritto

Programmazione Mobile
1. Mario Bianchi
2. Luigi Verdi

Tecnologie Web
1. Mario Bianchi
2. Luigi Verdi
3. Giovanni Rossi

Cloud and Edge Computing
1. Mario Bianchi
2. Luigi Verdi
3. Giovanni Rossi
4. Maria Viola

Internet, Web e Cloud
1. Mario Bianchi
2. Luigi Verdi
3. Giovanni Rossi
4. Maria Viola
5. Antonia Azzurri

Bonus

Combinare Views ereditare da `django.views.generic.*` con input arbitrari passati tramite url path.

#ricerca per cognome degli studenti (iscrizioni/urls.py):

```
path("studente/<str:surname>/",  
     views.ListStudenteBySurname.as_view(),  
     name="studente")
```

```
# Bonus: ListView in grado di prendere in ingresso un cognome
# passato come url path i.e. studenti/<str:surname>/ ed elencare
# gli studenti con tale cognome

class ListStudianteBySurname(ListaStudentiView):

    #da ListaStudentiView andiamo ad ereditare gli attributi model e template_name

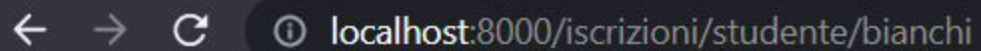
    def get_queryset(self):

        arg = self.kwargs["surname"] #leggiamo l'argomento passato
        qs = self.model.objects.filter(surname__iexact=arg)

        #iexact significa case insensitive...

        return qs
```

Risultato



← → ↻ ⓘ localhost:8000/iscrizioni/studente/bianchi

ID: 66: Mario Bianchi

Esercizio Complesso

Creare un url (e rispettiva logica) che porti ad una pagina raggiungibile tramite “cercastudente/” in cui compare un form in cui l’utente può fare una ricerca per studente tramite nome **e/o** cognome. Al click del pulsante “ricerca”, i dati spediti devono essere mandati tramite metodo **post**.

Si operi quindi una **redirezione** su una list View degli studenti che soddisfano i criteri di ricerca. Si visualizzino inoltre i corsi a cui sono iscritti.

```
def cerca_studenti(request):  
    if request.method == "GET":  
        return render(request,template_name="iscrizioni/cerca_studenti.html")  
    else:  
        if len(request.POST["name"]) < 1:  
            nome = "null"  
        else: nome = request.POST["name"]  
        if len(request.POST["surname"]) < 1:  
            cognome = "null"  
        else: cognome = request.POST["surname"]  
        return redirect("iscrizioni:studentecercato",name=nome, surname=cognome)
```

```
class ListStudenteByNameAndSurname(ListView):

    model = Studente
    template_name = "iscrizioni/listastudenteinsegnamento.html"

    def get_queryset(self):

        try:
            arg = self.kwargs["name"]
            qs_name = self.model.objects.filter(name__iexact=arg)
        except:
            qs_name = self.model.objects.none()

        try:
            arg = self.kwargs["surname"]
            qs_surname = self.model.objects.filter(surname__iexact=arg)
        except:
            qs_surname = self.model.objects.none()

        return (qs_name | qs_surname)

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['titolo'] = "Studenti e loro insegnamenti"
        ls = set()
        for s in self.get_queryset():
            for i in Insegnamento.objects.all():
                if s in i.studenti.all():
                    ls.add(i)

        context['set_ins'] = ls

        return context
```

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 %}
```

listastudenteinsegnamento.html

```
{% block content %}

<h1>  {{ titolo }} </h1>

{% if object_list.count == 0 %}

<p> La ricerca non ha portato a risultati. </p>

{% else %}

    {% for p in object_list %}

        <ul> {{ p }} </ul>

    {% endfor %}

    <p> Iscrizioni ai corsi </p>

    {% for i in set_ins %}

        <ul> <a href="{% url 'iscrizioni:insegnamento' pk=i.pk %}">{{ i }}</a></ul>

    {% endfor %}

{% endif %}
```