Menu

Contents

[1. Cài đặt 4](#_Toc531379575)

[2. Django – Tạo project Django 4](#_Toc531379576)

[Tạo project 4](#_Toc531379577)

[Run server 4](#_Toc531379578)

[Tạo web app 4](#_Toc531379579)

[Tạo view 5](#_Toc531379580)

[3. Thiết lập cơ sở dữ liệu 7](#_Toc531379581)

[Tạo mô hình dữ liệu cho ứng dụng 8](#_Toc531379582)

[Tạo bảng từ mô hình dữ liệu 9](#_Toc531379583)

[Thao tác với các bảng 10](#_Toc531379584)

[4. Backend System 12](#_Toc531379585)

[Tạo user 12](#_Toc531379586)

[5. View và Template 13](#_Toc531379587)

[Tạo view 13](#_Toc531379588)

[Template 14](#_Toc531379589)

[Xử lý cho lỗi 404 16](#_Toc531379590)

[URL động 17](#_Toc531379591)

[Namespace cho URL 17](#_Toc531379592)

[6. Form trong Django 18](#_Toc531379593)

[Tạo form trong template 19](#_Toc531379594)

[Tạo Form class kế thừa từ Form 21](#_Toc531379595)

[Tạo Form class kế thừa từ ModelForm 21](#_Toc531379596)

[7. File tĩnh 21](#_Toc531379597)

[Tùy biến CSS 22](#_Toc531379598)

[8. Tùy chỉnh trang Admin 22](#_Toc531379599)

[Tùy chỉnh form thêm / sửa 22](#_Toc531379600)

[Tùy chỉnh model chứa khóa ngoại 24](#_Toc531379601)

[Tùy chỉnh trang danh sách 27](#_Toc531379602)

[Ví dụ khác về chỉnh sửa trong admin 29](#_Toc531379603)

[9. Working with QuerySet and managers 30](#_Toc531379604)

[Creating objects 30](#_Toc531379605)

[Updating objects 31](#_Toc531379606)

[Retrieving objects 31](#_Toc531379607)

[Using the filter() method 31](#_Toc531379608)

[Using exclude() 31](#_Toc531379609)

[Using order\_by() 31](#_Toc531379610)

[Deleting objects 31](#_Toc531379611)

[Creating model managers 32](#_Toc531379612)

[10. Gởi mail trong Django 32](#_Toc531379613)

[11. Quan hệ giữa các model trong Django 35](#_Toc531379614)

[Quan hệ 1-n 35](#_Toc531379615)

[Quan hệ 1-1 36](#_Toc531379616)

[Quan hệ n-n 36](#_Toc531379617)

[12. Sử dụng thư viện bên thứ 3 36](#_Toc531379618)

[Sử dụng taggit cho Blog 37](#_Toc531379619)

[13. Using the Django authentication framework 39](#_Toc531379620)

[Tạo login view 39](#_Toc531379621)

[Using Django authentication views 39](#_Toc531379622)

[Log in and log out views 40](#_Toc531379623)

[Create a new view to display a dashboard 41](#_Toc531379624)

[Thay đổi password user 42](#_Toc531379625)

[Reset password user 42](#_Toc531379626)

[14. Using the messages framework 43](#_Toc531379627)

[15. Study Case: Buid social web 45](#_Toc531379628)

[16. Django nâng cao 45](#_Toc531379629)

[Sử dụng Generic relation foreign key 45](#_Toc531379630)

[Creating custom model fields 45](#_Toc531379631)

[Using class-based views 46](#_Toc531379632)

[{% include "pagination.html" with **page=page\_obj** %} 47](#_Toc531379633)

[Sử dụng Mixins 47](#_Toc531379634)

[Sử dụng Django access mixin 49](#_Toc531379635)

[Sử dụng formset 50](#_Toc531379636)

[17. Login và logout sử dụng view của Django 50](#_Toc531379637)

[18. How to Implement Multiple User Types with Django 53](#_Toc531379638)

[19. Gặp lỗi khi gọi lệnh: python manage.py makemigrations 70](#_Toc531379639)

[20. get\_context\_data() 71](#_Toc531379640)

[21. Django CRUD (Create, Retrieve, Update, Delete) 71](#_Toc531379641)

[Install Django and Start New Project 72](#_Toc531379642)

[Create new App 72](#_Toc531379643)

[Create the Model 72](#_Toc531379644)

[Admin Interface (optional) 73](#_Toc531379645)

[The Views 73](#_Toc531379646)

[Define the URLs 73](#_Toc531379647)

[Templates 74](#_Toc531379648)

[Test It 75](#_Toc531379649)

[Function Based View Version 75](#_Toc531379650)

# Cài đặt

Lệnh cài đặt

pip install Django

Nếu cài theo phiên bản

pip install Django==1.9.4

Kiểm tra phiên bản

Code

import django

print(django.get\_version())

# Django – Tạo project Django

## Tạo project

Lệnh tạo

django-admin startproject mysite

Ở đây mysite là project web cần tạo. Cấu trúc thư mục lúc này sẽ như sau:

mysite/

    manage.py

    mysite/

        \_\_init\_\_.py

        settings.py

        urls.py

        wsgi.py

Mỗi ứng dụng web Django sẽ chạy riêng 1 server nên có thể đặt code web ở bất kì đâu (khác với PHP, JSP… cần cài webserver)

## Run server

Lệnh chạy

python manage.py runserver

manage.py là file cấu hình server cho mỗi ứng dụng web. Mặc định ứng dụng web python sẽ chạy trên cổng 8000.

Để đổi cổng chạy sang cổng khác (vd 8080) thực hiện như sau:

python manage.py runserver 8080

## Tạo web app

Phần trên là tạo webserver cho website, để tạo ứng dụng web, thực hiện:

python manage.py startapp polls

Với polls là ứng dụng cần tạo ra

Cấu trúc ứng dụng web sẽ như sau:

polls/

    \_\_init\_\_.py

    admin.py

    apps.py

    migrations/

        \_\_init\_\_.py

    models.py

    tests.py

    views.py

Project web khác với app web là project chứa nhiều app, mỗi app thực hiện một chức năng, công việc riêng biệt.

## Tạo view

polls/views.py

from django.http import HttpResponse

def index(request):

    response = HttpResponse()

    response.write("<h1>Welcome</h1>")

    response.write("This is the polls app")

    return response

Bên trong file views.py này chúng ta viết hàm index, hàm này trả về một đối tượng HttpResponse.

Để gọi được tới hàm index này và lấy nội dung HTML thì chúng ta phải tạo đường dẫn URL tới file này.

polls/

    \_\_init\_\_.py

    admin.py

    apps.py

    migrations/

        \_\_init\_\_.py

    models.py

    tests.py

    urls.py

    views.py

Để tạo URL chúng ta tạo một file với tên urls.py bên trong thư mục polls.

polls/urls.py

from django.conf.urls import url

from . import views

 urlpatterns = [

     url(r'^$', views.index, name='index'),

]

Django sẽ tìm thông tin về các url trong list urlpatterns, list này chứa các đối tượng url, mỗi url bao gồm 3 tham số :

- tham số thứ nhất là biểu thức chính quy (Regex – Regular Expression),

- tham số thứ hai là tên module và tên hàm sẽ trả về nội dung HTML, ở đây là hàm index trong module views (file views.py),

- tham số thứ 3 là tên biến toàn cục trong toàn bộ ứng dụng, trong đó 2 tham số đầu tiên là bắt buộc phải có.

Tiếp theo chúng ta phải tạo đường dẫn đến ứng dụng polls này, chúng ta sẽ tạo đường dẫn cho từng ứng dụng trong file mysite/urls.py.

mysite/urls.py

from django.conf.urls import include, url

from django.contrib import admin

urlpatterns = [

    url(r'^polls/', include('polls.urls')),

    url(r'^admin/', admin.site.urls),

]

Bên trong file mysite/urls.py cũng có một list urlpatterns, list này chứa các đối tượng url dẫn đến các ứng dụng web khác mà chúng ta viết.

Ở đây tham số thứ 2 chúng ta không chỉ đến một hàm index nào đó trong một module nào đó, mà chúng ta chỉ đến một module có chứa đối tượng urlpatterns khác, trong trường  hợp này là polls.urls, nên chúng ta phải dùng đến hàm include(), đối với các url chỉ đến một module url khác thì chuỗi regex không kết thúc bằng dấu $, hay nói cách khác là không có kết thúc.

Khi chúng ta gõ localhost:8000 thì django sẽ đọc các url trong file mysite/urls.py, trong file này chúng ta có một đường dẫn đến admin/, đường dẫn này trỏ đến hàm get\_url() trong module admin.site của django, do đó bạn có thể gõ localhost:8000/admin và django sẽ trả về một trang đăng nhập mẫu (chúng ta sẽ tìm hiểu về trang admin này sau), url thứ hai là đường dẫn polls/ trỏ đến một file urls.py khác ở module polls.urls do chúng ta tự viết, file này lại chứa một list urlpatterns khác chứa các đường dẫn đến các hàm trả về HTML riêng của chúng, thế nên chúng ta có thể gõ localhost:8000/polls, và django sẽ trả về trang HTML tạo ra từ hàm index() trong module polls.urls.

# Thiết lập cơ sở dữ liệu

Mặc định là SQLite. Nếu thay đổi vào file settings.py trong myweb để thay đổi.

DATABASES = {

'default': {

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

'NAME': os.path.join(BASE\_DIR, 'db.sqlite3'),

}

}

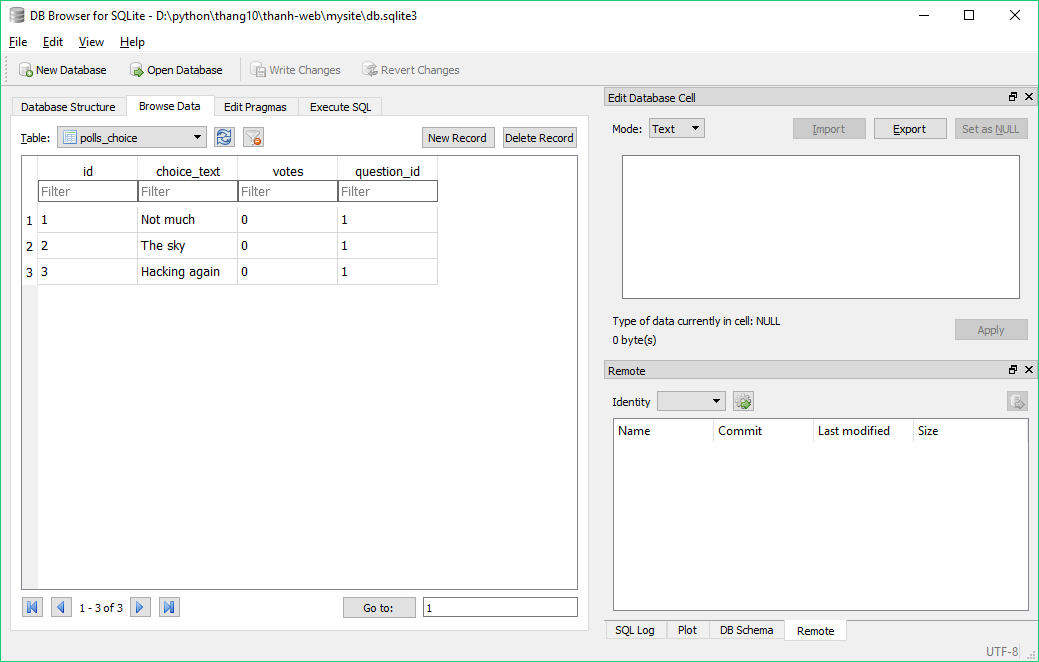
* ENGINE: tên module dành cho từng CSDL, mặc định sử dụng SQLite
  + django.db.backends.sqlite3 – cơ sở dữ liệu  SQLite
  + django.db.backends.postgresql – cơ sở dữ liệu PostgreSQL
  + django.db.backends.mysql – cơ sở dữ liệu MySQL
  + django.db.backends.oracle – cơ sở dữ liệu Oracle
* NAME: tên CSDL, mặc định là file db.sqlite3 được tạo ra ở thư mục gốc của server.

List INSTALLED\_APPS là các ứng dụng thường được sử dụng khi chạy web. Lệnh:

python manage.py migrate

sẽ tìm các module trong list này và tạo các table tương ứng.

Để quản lý CSDL SQLite có thể sử dụng add-on của Firefox, Chrome hoặc sử dụng chương trình như DB Browser for SQLite (sqlitebrowser.org).



## Tạo mô hình dữ liệu cho ứng dụng

polls/models.py

from django.db import models

class Question(models.Model):

    question\_text = models.CharField(max\_length=200)

    pub\_date = models.DateTimeField('date published')

class Choice(models.Model):

    question = models.ForeignKey(Question, on\_delete=models.CASCADE)

    choice\_text = models.CharField(max\_length=200)

    votes = models.IntegerField(default=0)

Mỗi mô hình được tạo ra phải thừa kế từ lớp: django.db.models.Model

Ví dụ khác về mô hình dữ liệu cho Blog

from django.db import models

from django.utils import timezone

from django.contrib.auth.models import User

# Create your models here.

class Post(models.Model):

STATUS\_CHOICES = (

('draft', 'Draft'),

('published', 'published')

)

title = models.CharField(max\_length=250)

slug = models.SlugField(max\_length=250, unique\_for\_date='publish')

author = models.ForeignKey(User, related\_name='blog\_posts')

body = models.TextField()

publish = models.DateTimeField(default=timezone.now)

# the date will be saved automatically when creating an object

created = models.DateTimeField(auto\_now\_add=True)

# the date will be updated automatically when saving an object

updated = models.DateTimeField(auto\_now=True)

status = models.CharField(max\_length=10, choices = STATUS\_CHOICES, default='draft')

# The class Meta inside the model contains metadata

class Meta:

#descending order by using the negative prefix

ordering = ('-publish', )

#The \_\_str\_\_() method is the default human-readable representation of the object.

def \_\_str\_\_(self):

return self.title

Lưu ý: thuộc tính ordering trong class Meta phải có dấu ‘,’ nếu chỉ có 1 thuộc tính, nếu không sẽ bị lỗi.

## Tạo bảng từ mô hình dữ liệu

Sau khi đã tạo các lớp mô hình trong Python, Django sẽ nhìn các thuộc tính trong từng **lớp để tạo các bảng** tương ứng trong CSDL và **tạo thêm các lớp mới** trong Python cung cấp các hàm để thao tác với các bảng trong CSDL.

|  |
| --- |
| mysite/settings.py |
|  |

INSTALLED\_APPS = [

'polls.apps.PollsConfig',

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

]

Để Django tạo các bảng mới trong CSDL thì chúng ta phải khai báo trước trong list INSTALLED\_APPS đã bằng cách thêm dòng polls.apps.PollsConfig.

|  |
| --- |
|  |

Tiếp theo chạy lệnh makemigration polls để báo cho Django biết là bạn đã thay đổi một số mô hình, ở đây là thêm 2 lớp mới trong gói polls.

python manage.py makemigrations polls

Lệnh migrate để Django cập nhật lại CSDL. Lúc này CSDL chúng ta sẽ có thêm 2 bảng mới là polls\_choice và polls\_question (tên bảng được đặt theo cú pháp <tên package>\_<tên lớp>).

python manage.py migrate

## Thao tác với các bảng

Sau khi Django tạo các bảng, ta có thể thao tác các công việc thường dùng như thêm-xóa-sửa. Sử dụng shell của python thao tác với csdl, chạy lệnh:

python manage.py shell

Phương thức Question.objects.all() liệt kê toàn bộ đối tượng Question đang có trong CSDL

>>> from polls.models import Question, Choice

>>> Question.objects.all()

[]

Tạo một đối tượng Question và thiết lập pub\_date là ngày giờ hiện tại của máy tính

>>>from django.utils import timezone

>>>q = Question(question\_text="What's new?", pub\_date=timezone.now())

>>>q.save()

Khi viết lớp Question và Choice chúng ta không cung cấp thuộc tính id hay bất cứ thuộc tính nào để làm khóa chính, vì vậy khi tạo CSDL Django sẽ tự động thêm 1 thuộc tính id để làm khóa chính và thuộc tính này sẽ tự động tăng.

>>> q.id

1

Ta có thể thay đổi trực tiếp giá trị trên các đối tượng này sau đó gọi phương thức save() là dữ liệu trên CSDL sẽ được cập nhật

>>> q.question\_text

What's new?

>>> q.question\_text = "What's up?"

>>> q.save()

Nên override phương thức \_\_str\_\_() để khi in đối tượng ra người dùng có thể đọc được. Nếu không override phương thức này thì Question.objects.all() sẽ cho ra kết quả không đọc được.

polls/models.py

from django.db import models

class Question(models.Model):

    question\_text = models.CharField(max\_length=200)

    pub\_date = models.DateTimeField('date published')

    def \_\_str\_\_(self):

        return self.question\_text

class Choice(models.Model):

    question = models.ForeignKey(Question, on\_delete=models.CASCADE)

    choice\_text = models.CharField(max\_length=200)

    def \_\_str\_\_(self):

        return self.choice\_text

Chạy kết quả khi có hàm \_\_str\_\_. Lúc này phương thức objects.all() sẽ gọi đến phương thức \_\_str\_\_().

>>> from polls.models import Question, Choice

>>> Question.objects.all()

[<Question: What's up?>]

Lọc các bản ghi theo thuộc tính bằng phương thức filter().

>>> Question.objects.filter(id=1)

[<Question: What's up?>]

>>> Question.objects.filter(question\_text\_\_startswith='What')

[<Question: What's up?>]

Ta cũng có thể lấy từng bản ghi đơn lẻ bằng phương thức get(), ở trên chúng ta lấy theo khóa chính pk (Primary Key).

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

Bất cứ bảng nào có một bảng khác chứa khóa ngoại tham chiếu đến khóa chính của nó đều sẽ có một thuộc tính được tạo tự động là một tập hợp các đối tượng của bảng kia. Ở đây bảng Choice chứa khóa ngoại tham chiếu đến bảng Question, do đó bảng Question sẽ có một danh sách các đối tượng Choice, chúng ta có thể lấy danh sách này qua choice\_set.all() (tên danh sách được đặt theo <tên bảng>\_set). Ở đây chúng ta vẫn chưa tạo đối tượng Choice nào nên danh sách trả về rỗng.

>>> q.choice\_set.all()

[]

Chúng ta tạo 3 đối tượng Choice bằng phương thức choice\_set.create().

>>> q.choice\_set.create(choice\_text='Not much', votes=0)

<Choice: Not much>

>>> q.choice\_set.create(choice\_text='The sky', votes=0)

<Choice: The sky>

>>> c = q.choice\_set.create(choice\_text='Just hacking again', votes=0)

Chúng ta có thể lấy số lượng các bản ghi trong bảng bằng phương thức count().

>>> q.choice\_set.all()

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

>>> q.choice\_set.count()

3

Để xóa một bản ghi trong bảng thì chúng ta dùng phương thức delete().

>>> c = q.choice\_set.filter(choice\_text\_\_startswith='Just hacking')

>>> c.delete()

# Backend System

## Tạo user

Tạo tài khoản admin sử dụng lệnh:

python manage.py createsuperuser

Điền thông tin cần thiết như username, email, password

Username: admin

Email address: admin@example.com

Password:

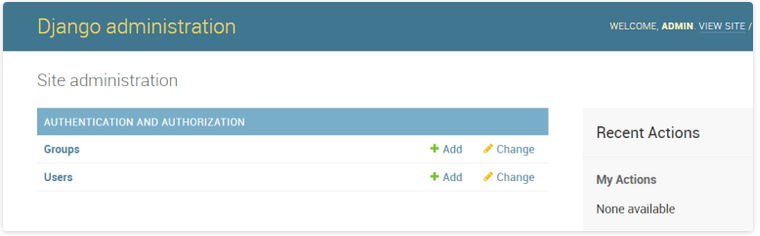
Password (again):

Superuser created successfully.

Sử dụng tài khoản vừa tạo đăng nhập vào trang quản trị với đường dẫn:

<http://127.0.0.1:8000/admin>

Trang quản trị sau khi đăng nhập thành công



Để đăng ký các bảng (hay các mô hình) với admin, sử dụng phương thức admin.site.register() trong file admin.py mà Django đã tạo cho chúng ta.

polls/admin.py

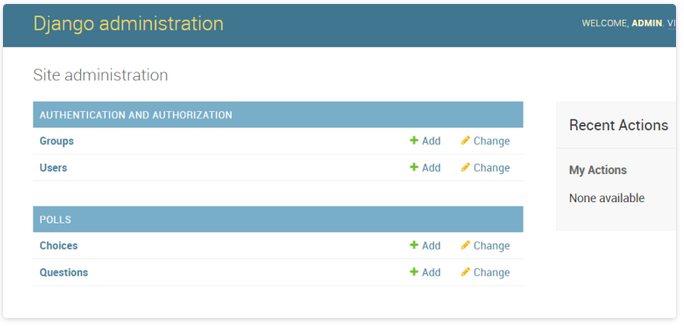
from django.contrib import admin

# Register your models here.

from .models import Question, Choice

admin.site.register(Question)

admin.site.register(Choice)



# View và Template

Trong Django thì một View là **một hàm/phương thức** làm một công việc cụ thể nào đó, một View  thường đi kèm với một Template.

Ví dụ, một ứng dụng Blog có một số View sau:

* Home – hiển thị các bài viết mới nhất.
* Entry – hiển thị bài viết nào đó.
* Archive – Lưu trữ các bài viết theo năm/tháng.
* Comment – xử lý việc đăng bình luận của độc giả.

Trong Django, một trang web được tạo ra bởi các hàm View, Django sẽ chọn View nào tùy thuộc vào URL mà chúng ta đã thiết lập.

## Tạo view

Thêm 3 view mới cho ứng dụng polls:

polls/view.py

def detail(request, question\_id):

    return HttpResponse("You're looking at question %s." % question\_id)

def results(request, question\_id):

    response = "You're looking at the results of question %s."

    return HttpResponse(response % question\_id)

def vote(request, question\_id):

    return HttpResponse("You're voting on question %s." % question\_id)

Tạo URL trỏ đến view

polls/urls.py

from django.conf.urls import url

from . import views

urlpatterns = [

    # ex: /polls/

    url(r'^$', views.index, name='index'),

    # ex: /polls/5/

    url(r'^(?P<question\_id>[0-9]+)/$', views.detail, name='detail'),

    # ex: /polls/5/results/

    url(r'^(?P<question\_id>[0-9]+)/results/$', views.results, name='results'),

    # ex: /polls/5/vote/

    url(r'^(?P<question\_id>[0-9]+)/vote/$', views.vote, name='vote'),

Ví dụ: localhost:8000/polls/34/, Django sẽ gọi đến hàm detail()

Khi gõ địa chỉ lên thanh URL của trình duyệt:

- Django sẽ đọc biến urlpatterns trong file mysite/settings.py, các đối tượng url sẽ được đọc dần dần từ trên xuống dưới cho đến khi có một đường dẫn vừa khít với URL được nhập vào.

- Khi tìm thấy localhost:8000/polls vừa khít với '^polls/', Django sẽ cắt đoạn phía sau URL ra (/polls/34/) và gửi đến file polls/urls.py để tiếp tục quá trình tìm kiếm

- Chuỗi (/polls/34/) vừa khít Regex r'^(?P<question\_id>[0-9]+)/$', => phương thức được gọi views.detail()với tham số như sau:

detail(request=<HttpRequest object>, question\_id='34')

Chuỗi Regex (?P<question\_id>[0-9]+) cho Django biết có một chuỗi con trong đoạn URL có dạng một con số (có 1 hoặc nhiều chữ số) và biến được truyền vào phương thức detail() có tên là question\_id.

## Template

Các hàm View trả về một trong 2 thứ: hoặc là trả về một đối tượng HttpReponse chứa nội dung HTML để hiển thị lên trình duyệt, hoặc là một lỗi exception 404.

Sửa lại hàm index() như sau:

polls/views.py

from django.http import HttpResponse

from .models import Question

def index(request):

    latest\_question\_list = Question.objects.order\_by('-pub\_date')[:5]

    output = ', '.join([q.question\_text for q in latest\_question\_list])

    return HttpResponse(output)

Đoạn code trên lấy ra 5 đối tượng Question được thêm vào gần đây nhất, sau đó lấy thuộc tính question\_text đưa vào biến output rồi trả về.

Khi chạy với địa chỉ localhost:8000/polls/ thì dữ liệu sẽ in ra với định dạng thô, chưa được định dạng. Ta có thể định dạng dữ liệu ngay trong view bằng các thẻ hay CSS, tuy nhiên khó quản lý và code lộn xộn.

Django cung cấp cho chúng ta hệ thống Template, chúng ta có thể viết phần giao diện ở Template và dùng nó để hiển thị dữ liệu của các View cho đẹp mắt hơn, mục đích chính của View chính là kết nối giữa Template và Model.

Đầu tiên chúng ta tạo một thư mục có tên là templates trong thư mục polls, Django sẽ tự động tìm các file template trong thư mục này. Ở đây bạn tạo thêm một thư mục khác nữa tên là tem1, trong thư mục này bạn tạo một file tên index.html.

polls/templates/temp1/index.html

{% if latest\_question\_list %}

    <ul>

       {% for question in latest\_question\_list %}

       <li>

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

                {{question.question\_text}}

             </a>

       <li>

       {% endfor %}

    </ul>

{% else %}

   <p>No polls are available.</p>

{% endif %}

Thẻ template bắt đầu và kết thúc bằng cặp kí tự {% %} hoặc {{ }}, các câu lệnh Python nằm trong cặp dấu {% %}, còn các biến thì nằm trong cặp {{ }}.

Sửa lại đoạn code trong hàm index() để sử dụng template.

polls/views.py

#để sử dụng template

**from django.shortcuts import render**

from django.http import HttpResponse

from django.template import loader

from .models import Question

def index(request):

    latest\_question\_list = Question.objects.order\_by('-pub\_date')[:5]

#template sử dụng (từ thư mục templates)

**template = loader.get\_template('tem1/index.html')**

**context** = {

        'latest\_question\_list': latest\_question\_list,

    }

    return HttpResponse(**template.render(context, request)**)

Tham số đầu tiên là nội dung HTML trả về, tham số thứ 2 là đối tượng request được gửi đến.

Hoặc cũng có thể trả về template trực tiếp trong hàm render() như sau

polls/views.py

from django.shortcuts import render

def index(request):

    latest\_question\_list = Question.objects.order\_by('-pub\_date')[:5]

    context = {

        'latest\_question\_list': latest\_question\_list,

    }

    return **render(request, 'temp1/index.html', context)**

Với tham số thứ nhất là đối tượng request, tham số thứ 2 là đường dẫn đến file template, tham số thứ 3 là nội dung HTML trả về.

## Xử lý cho lỗi 404

polls/views.py

from django.shortcuts import render

**from django.http import HttpResponse, Http404**

from django.template import loader

from .models import Question

def detail(request, question\_id):

**try:**

        question = Question.objects.get(pk=question\_id)

**except Question.DoesNotExist:**

**raise Http404("Question does not exist")**

    return render(request, 'tem1/detail.html', {'question':question})

Phương thức get() sẽ trả về một exception DoesNotExist nếu không tìm thấy bản ghi nào trong list, tại đây chúng ta giải phóng một đối tượng Http404.

polls/templates/temp1/detail.html

<h1>{{ question.question\_text }}</h1>

<ul>

    {% for choice in question.choice\_set.all %}

        <li>{{ choice.choice\_set }}</li>

    {% endfor %}

</ul>

## URL động

Trong các file template như index.html, các đường link chúng ta tham chiếu đến trong các thẻ <a> là do chúng ta tự viết

<li>

    <a href="/tem1/{{ question.id }}/">

        {{ question.question\_text }}

    </a>

</li>

Giả sử trong file template có hàng trăm thẻ a như vậy mỗi lần phải viết đầy đủ đường dẫn href="/tem1/…. và nếu thay đổi đường dẫn giả sử tem1 thành temp/abc/tem2 thì phải thay thủ công rất mất thời gian. Cách giải quyết hiệu quả là sử dụng tham số name trong url.

Khi chúng ta tạo các đối tượng url trong file urls.py, tham số thứ 3 là name, tham số này do chúng ta tự đặt, và Django cho phép chúng ta tham chiếu đến chúng trong các file template -dùng tên này thay cho đường dẫn url cố định.

<a href="{% url 'detail' question.id %}">

    {{ question.question\_text }}

</a>

Khi nào cần thay đổi URL mới thì chúng ta chỉ cần thay đường dẫn tham số đầu của url trong file urls.py là được:

polls/urls.py

...

url(r'^details/(?P<question\_id>[0-9]+)/$', views.detail, name='detail'),

...

## Namespace cho URL

Khi dùng đến URL động thì lại phát sinh vấn đề giả sử khi chúng ta có thêm nhiều ứng dụng khác ngoài polls, chẳng hạn như một ứng dụng blog, trong đó cũng có hàm view detail(), và hàm view này cũng sử dụng một template tên là detail.html

Giả sử sử dụng đường dẫn như sau:

<a href="{% url 'detail' question.id %}">

    {{ question.question\_text }}

</a>

Lúc này Django không phân biệt được của ứng dụng nào => không lấy được đường dẫn mong muốn.

namespace sẽ giúp giải quyết vấn đề này url

polls/urls.py

from django.conf.urls import url

from . import views

**app\_name = "polls"**

urlpatterns = [

    # ex: /polls/

    url(r'^$', views.index, name='index'),

    # ex: /polls/detils/5/

    #url(r'^(?P<question\_id>[0-9]+)/$', views.detail, name='detail'),

    url(r'^details/(?P<question\_id>[0-9]+)/$', views.detail, name='detail'),

    # ex: /polls/5/results/

    url(r'^(?P<question\_id>[0-9]+)/results/$', views.results, name='results'),

    # ex: /polls/5/vote/

    url(r'^(?P<question\_id>[0-9]+)/vote/$', views.vote, name='vote')

]

Để đặt tên namespace cho các đối tượng url thì chúng ta chỉ cần đặt giá trị cho biến app\_name trong file urls.py là được.

Trong file template, vd như index.html thì chúng ta chỉ cần viết đủ tên <namespace>:<tên\_biến\_url> là xong, vd url có name là detail thì viết đầy đủ thành polls:detail.

polls/templates/tem1/index.html

<a href="{% url 'polls:detail' question.id %}">

    {{ question.question\_text }}

</a>

# Form trong Django

Có nhiều cách tạo form trong Django:

- Tạo form trong template

- Tạo FormClass trong file forms.py kế thừa từ Form và load form này vào file template

- Tạo FormClass trong file forms.py kế thừa từ ModelForm và load form này vào file template

## Tạo form trong template

Thay đổi template detail.html

polls/templates/tem1/detail.html

<h1>{{ question.question\_text }}</h1>

{% if error\_messsage %}

<strong>{{ error\_messsage }}</strong>

{% endif %}

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

    {% csrf\_token %}

    {% for choice in question.choice\_set.all %}

        <input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}" />

        <label for="choice{{ forloop.counter }}">{{ choice.choice\_text }}</label>

    {% endfor %}

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

</form>

Tạo thẻ form và đưa url dẫn đến view vote() với tham số question.id là ID của đối tượng Choice trong CSDL, phương thức gửi lên là phương thức POST.

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

…

</form>

Hiển thị các radio button, mỗi radio button hiển thị một câu trả lời (Choice) cho từng Question. Các radio button sẽ được gán thuộc tính name là choice. Thuộc tính forloop.counter là số thứ tự của vòng lặp giống như khi bạn viết for(int i...)

{% for choice in question.choice\_set.all %}

        <input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}" />

        <label for="choice{{ forloop.counter }}">{{ choice.choice\_text }}</label>

{% endfor %}

**hàm view vote() xử lý từng thao tác vote**

polls/views.py

from django.shortcuts import get\_object\_or\_404, render

from django.http import HttpResponse, Http404, HttpResponseRedirect

from django.core.urlresolvers import reverse

from .models import Question, Choice

def vote(request, question\_id):

    question = get\_object\_or\_404(Question, pk=question\_id)

    try:

        selected\_choice = question.choice\_set\_get(pk=request.POST['choice'])

    except (KeyError, Choice.DoesNotExist):

        return render(request, 'polls/detail.html', {

            'question': question,

            'error\_message': "You didn't select a choice.",

        })

    else:

        selected\_choice.votes += 1

        selected\_choice.save()

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

Đối tượng HttpResponseRedirect để tránh các trường hợp người dùng nhấn nút back trên trình duyệt (cũng có thể dùng dùng đối tượng HttpResponse)

polls/views.py

from django.shortcuts import get\_object\_or\_404, render

def results(request, question\_id):

    question = get\_object\_or\_404(Question, pk=question\_id)

    return render(request, 'polls/results.html', {'question': question})

polls/templates/temp1/results.html

<h1>{{ question.question\_text }}</h1>

<ul>

    {% for choice in question.choice\_set.all %}

    <li>

    {{ choice.choice\_text }} -- {{ choice.votes }} vote{{ choice.votes | pluralize }}

    </li>

    {% endfor %}

</ul>

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

## Tạo Form class kế thừa từ Form

Thường sử dụng cách này nếu form không sử dụng model. Ví dụ form để người dùng chia sẻ post không cần có model.

blog/forms.py

from django import forms

class EmailPostForm(forms.Form):

name = forms.CharField(max\_length=25)

email = forms.EmailField()

to = forms.EmailField()

comment = forms.CharField(required=False, widget=forms.Textarea)

Load form này vào template:

<h1>Share "{{post.title}}" by e-mail </h1>

<form action="." method="POST">

{{ form.as\_p }}

{% csrf\_token %}

<input type="submit" value="Send e-mail">

</form>

## Tạo Form class kế thừa từ ModelForm

Cách này thường được sử dụng để tạo Form động từ Model

from .models import Comment

class CommentForm(forms.ModelForm):

class Meta:

model = Comment #Comment o day la model

fields = ('name', 'email', 'body')

Load form này vào template

<h2>Add a new comment</h2>

<form action="." method="post">

{{ comment\_form.as\_p }}

{% csrf\_token %}

<p><input type="submit" value="Add comment"></p>

</form>

# File tĩnh

Ngoài nội dung HTML được sinh ra bởi server thì một ứng dụng web còn cần đến các file bổ sung khác, chẳng hạn như các file hình ảnh, Javascript, CSS… Trong Django thì các file này được gọi là file tĩnh.

Khai báo đường dẫn đến file tĩnh trong file mysite/settings.py

# Static files (CSS, JavaScript, Images)

# https://docs.djangoproject.com/en/1.9/howto/static-files/

STATIC\_URL = '/static/'

## Tùy biến CSS

Đầu tiên chúng ta tạo một thư mục có tên là static trong thư mục polls. Django sẽ tự động tìm các file tĩnh trong thư mục này giống như tìm các file template trong thư mục polls/templates vậy.

Trong thư mục static này chúng ta lại tạo một thư mục khác với tên là polls và tạo một file CSS có tên là style.css.

polls/static/polls/style.css

li a {

    color: green

}

Tiếp theo chúng ta cần sửa lại file index.html để load file style này

polls/templates/polls/index.html

{% load staticfiles %}

<link rel="stylesheet" type="text/css" href="{% static 'polls/style.css' %}" />

Dòng {% load staticfiles %} sẽ tự động gán đường dẫn đến thư mục mysite/polls/static/ vào một biến có tên static do Django tự đặt, từ đó bạn chỉ cần lấy biến static là có thể lấy được đường dẫn tuyệt đối đến thư mục này.

Sau đó chúng ta chỉ cần lấy biến static trong cặp thẻ {% %} ra và gắn thêm đường dẫn đến các file css, js… của bạn.

# Tùy chỉnh trang Admin

## Tùy chỉnh form thêm / sửa

Mặc định form sẽ hiển thị tất cả các field trong model lên form.

Để chọn chỉ một số thuộc tính cần hiển thị thực hiện thực hiện:

- Tạo lớp kế thừa từ ModelAdmin.

- Thuộc tính fields chứa các field cần sử dụng.

polls/admin.py

from django.contrib import admin

# Register your models here.

from .models import Question, Choice

# lớp kế thừa từ lớp admin.ModelAdmin

class QuestionAdmin(admin.ModelAdmin):

# tên các thuộc tính trong model mà chúng ta muốn hiện ra trong Admin

    fields = ['pub\_date', 'question\_text']

#truyền lớp này vào hàm admin.site.register()

admin.site.register(Question, QuestionAdmin)

admin.site.register(Choice)

Django còn cho phép gom nhóm các thuộc tính lại với nhau để hiển thị.

polls/admin.py

from django.contrib import admin

# Register your models here.

from .models import Question, Choice

class QuestionAdmin(admin.ModelAdmin):

    fieldsets = [

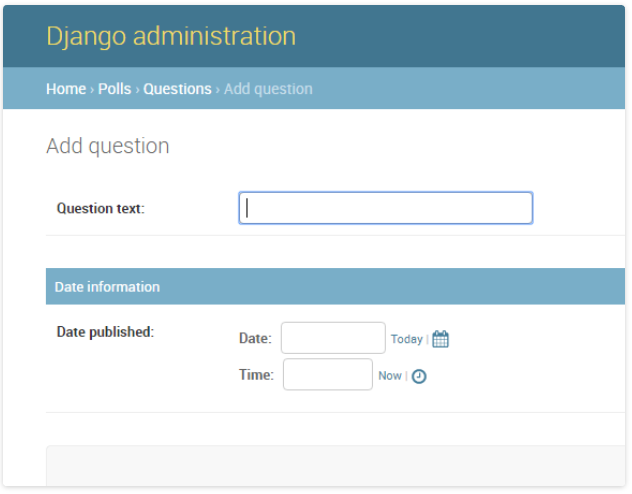
        (None,               {'fields': ['question\_text']}),

        ('Date information', {'fields': ['pub\_date']}),

    ]

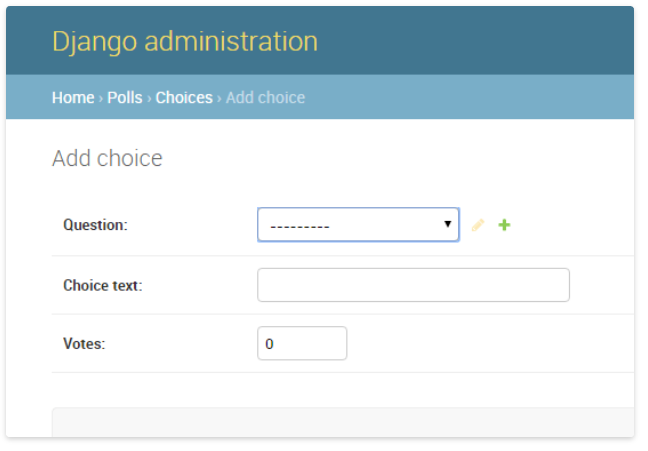
admin.site.register(Question, QuestionAdmin)

admin.site.register(Choice)



## Tùy chỉnh model chứa khóa ngoại

Mặc định giao diện trang tạo một đối tượng Choice mới như sau:



Chọn question xong chọn choice => không hợp lý, không phù hợp thực tế.

Ở đây sẽ tùy chỉnh trang để cho Choice hiển thị ngay bên dưới Question.

polls/admin.py

from django.contrib import admin

# Register your models here.

from .models import Question, Choice

**class ChoiceInLine(admin.StackedInline):**

**model = Choice # model su dung**

**extra = 3 #so luong form bao nhieu**

class QuestionAdmin(admin.ModelAdmin):

    fieldsets = [

        (None, {'fields': ['question\_text']}),

        ('Date information', {'fields': ['pub\_date']}),

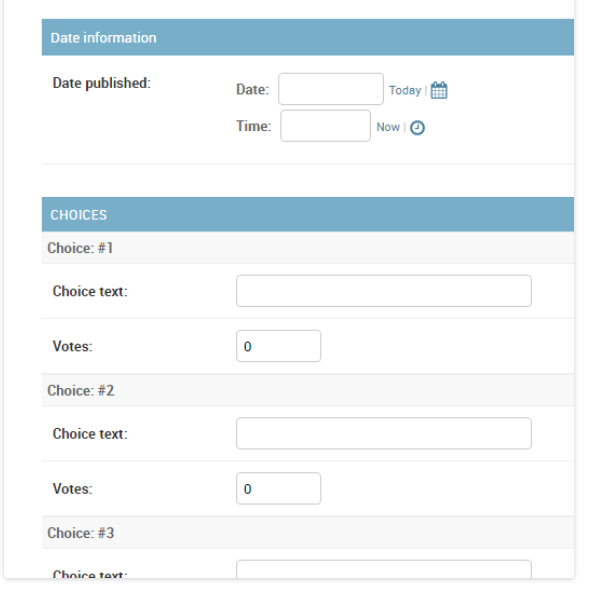
    ]

**inlines = [ChoiceInLine]**

admin.site.register(Question, QuestionAdmin)

admin.site.register(Choice)

Cần khai báo một lớp kế thừa từ lớp admin.StackedInLine, sau đó gán thuộc tính model giá trị là lớp Model mà chúng ta muốn hiện ra (Choice), thuộc tính extra cho Django biết có bao nhiêu form nhập vào được hiển thị khi hiển thị bảng cha. Sau đó trong lớp QuestionAdmin chúng ta khai báo thuộc tính inlines là lớp ChoiceInLine mà chúng ta vừa khai báo.



Có thể thay đổi kiểu hiển thị các đối tượng khóa ngoại từ dạng danh sách sang dạng bảng.

polls/admin.py

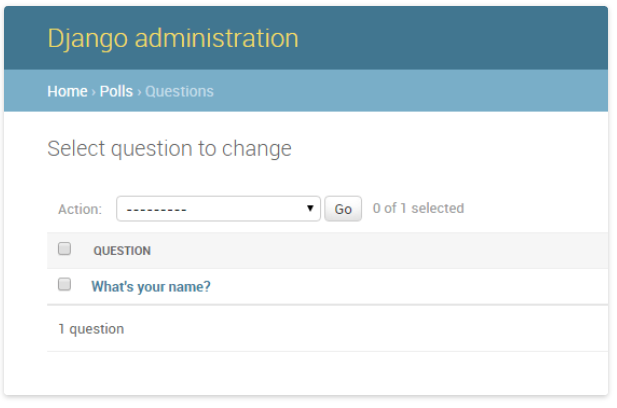
class ChoiceInLine(admin.**TabularInLine**):

    ...



## Tùy chỉnh trang danh sách

Hiện tại thì trang danh sách Question có giao diện như hình dưới.



Mặc định thì Django chỉ để hiển thị đoạn text lấy từ phương thức str() mà chúng ta đã override trong phương thức \_\_str\_\_() cho từng đối tượng, chúng ta có thể tùy chỉnh Django hiển thị những thuộc tính khác.

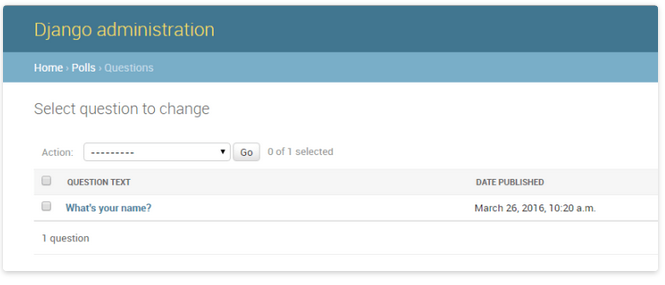
polls/admin.py

class QuestionAdmin(admin.ModelAdmin):

    ...

**list\_display = ('question\_text', 'pub\_date')**

Để làm việc này thì chúng ta chỉ cần khai báo danh sách các thuộc tính cần hiển thị ra trong thuộc tính list\_display.



Bạn thậm chí còn có thể sắp xếp các đối tượng Question bằng cách click chuột vào thanh tiêu đề.

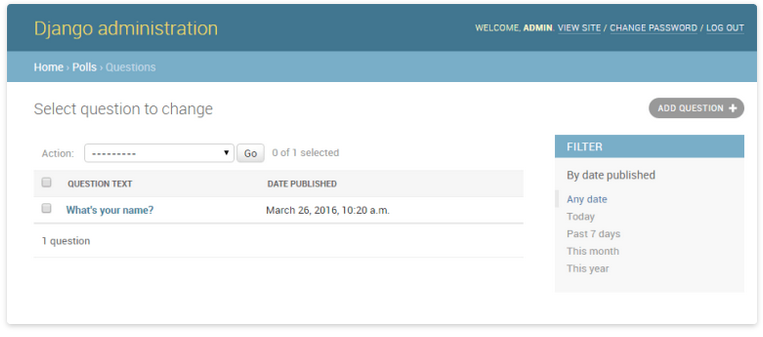
Ngoài ra bạn có thể thêm các ô lọc theo thuộc tính.

class QuestionAdmin(admin.ModelAdmin):

    ...

**list\_filter = ['pub\_date']**

Chúng ta khai báo tên các thuộc tính cần lọc trong thuộc tính list\_filter.



Thường filter đối với field kiểu là DateTime, ở đây là pub\_date, Django biết cách đưa các tùy chọn vào phần lọc chẳng hạn như *Any date*, *Today*…

polls/admin.py

class QuestionAdmin(admin.ModelAdmin):

    ...

**search\_fields = ['question\_text']**

Biến search\_fields sẽ hiển thị ô tìm kiếm và chỉ tìm theo thuộc tính mà chúng ta đã khai báo trong đó, bạn cũng có thể thêm các thuộc tính khác vào nhưng nên thêm ít thôi vì Django sử dụng câu truy vấn LIKE với CSDL nên thêm nhiều thuộc tính vào sẽ làm quá trình tìm kiếm chậm đi.

## Ví dụ khác về chỉnh sửa trong admin

blogs/admin.py

from django.contrib import admin

from .models import Post

# Register your models here.

class PostAdmin(admin.ModelAdmin):

list\_display = ('title', 'slug', 'author', 'publish', 'status')

list\_filter = ('status', 'created', 'publish', 'author')

search\_fields = ('title', 'body')

#prepopulate the slug field with the input of the title field

#using the prepopulated\_fields attribute

prepopulated\_fields = {'slug': ('title',)}

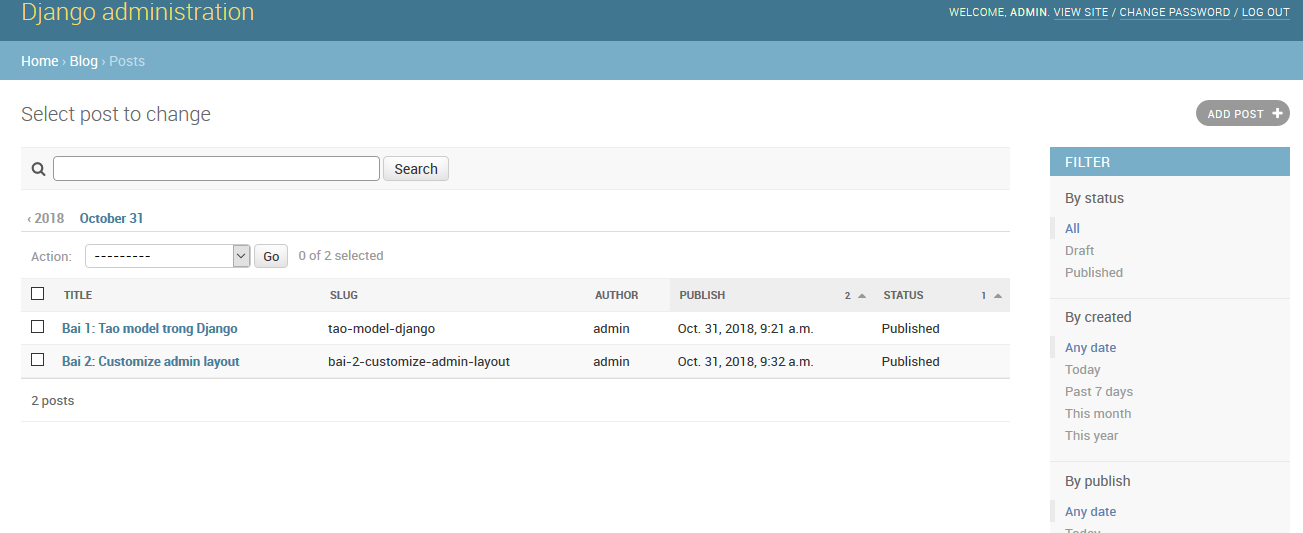
raw\_id\_fields = ('author',)

date\_hierarchy = 'publish'

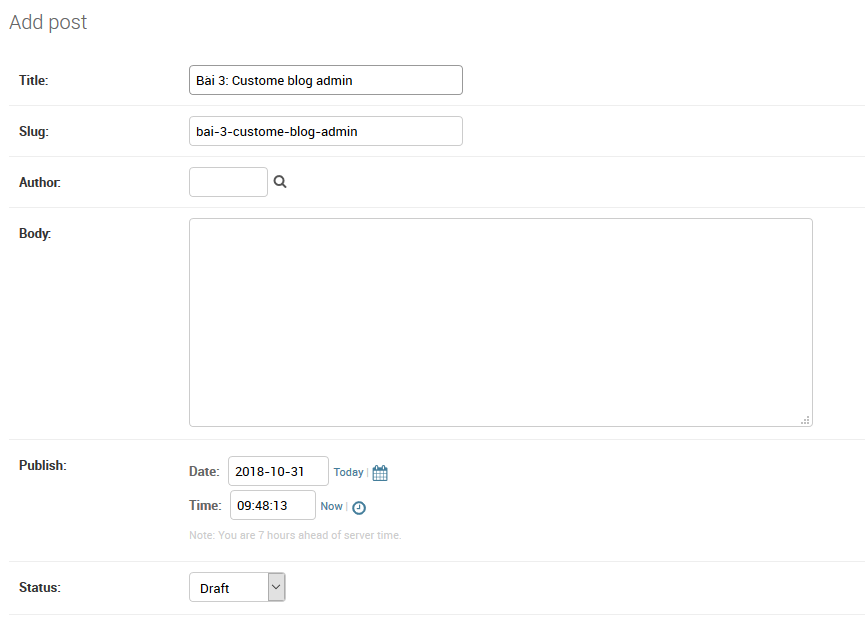
ordering = ['status', 'publish']

admin.site.register(Post, PostAdmin)

Giao diện sẽ như sau:



Giao diện thêm mới



Với 2 dòng dưới sẽ có tác dụng trong form thêm mới.

prepopulated\_fields = {'slug': ('title',)}

raw\_id\_fields = ('author',)

prepopulated\_fields: tự động điền dữ liệu cho slug với dữ liệu lấy từ title.

raw\_id\_fields giúp hiển thị kính lúp để tìm khóa ngoại.

# Working with QuerySet and managers

Django comes with a powerful database-abstraction API that lets you create, retrieve, update, and delete objects easily. The Django **Object-relational Mapper** (**ORM**) is compatible with MySQL, PostgreSQL, SQLite, and Oracle.

## Creating objects

from django.contrib.auth.models import User

from blog.models import Post

user = User.objects.get(username='admin')

Post.objects.create(title='One more post',

slug='one-more-post',

body='Post body.',

author=user)

post.save()

## Updating objects

post.title = 'New title'

post.save()

## Retrieving objects

The Django **Object-relational mapping** (**ORM**) is based on QuerySet.

Each Django model has at least one manager, and the default manager is called **objects**. You get a QuerySet object by using your models manager.

all\_posts = Post.objects.all()

user = User.objects.get(username='admin')

## Using the filter() method

To filter a QuerySet, you can use the filter() method of the manager.

Post.objects.filter(publish\_\_year=2015)

Post.objects.filter(publish\_\_year=2015, author\_\_username='admin')

Post.objects.filter(publish\_\_year=2015)\

filter(author\_\_username='admin')

Chú ý: We are building queries with field lookup methods using two underscores (publish\_\_year), but we are also accessing fields of related models using two underscores (author\_\_username).

## Using exclude()

You can exclude certain results from your QuerySet using the exclude() method of the manager. For example, we can retrieve all posts published in 2015 whose titles don't start by Why:

Post.objects.filter(publish\_\_year=2015)\

.exclude(title\_\_startswith='Why')

## Using order\_by()

You can retrieve all objects ordered by their title:

Post.objects.order\_by('title')

You can indicate descending order with a negative sign prefix, like this:

Post.objects.order\_by('-title')

## Deleting objects

If you want to delete an object, you can do it from the object instance:

post = Post.objects.get(id=1)

post.delete()

Note that deleting objects will also delete any dependent relationships.

## Creating model managers

**objects is t**he default manager of every model, which retrieves all objects in the database. But we can also define custom managers for our models. There are two ways to add managers to your models:

- add extra manager methods

- modify initial manager QuerySets.

The first one turns up something like Post.objects.my\_manager() and the later like Post.my\_manager.all(). Our manager will allow us to retrieve posts using Post.published.

blogs\models.py

class PublishedManager(models.Manager):

def get\_queryset(self):

return super(PublishedManager,

self).get\_queryset()\

.filter(status='published')

class Post(models.Model):

# ...

objects = models.Manager() # The default manager.

published = PublishedManager() # Our custom manager.

get\_queryset() is the method that returns the QuerySet to be executed. We use it to include our custom filter in the final QuerySet. We have defined our custom manager and added it to the Post model; we can now use it to perform queries. For example, we can retrieve all published posts whose title starts with Who using:

Post.published.filter(title\_\_startswith='Who')

# Gởi mail trong Django

Trong settings.py thêm các cài đặt:

EMAIL\_HOST: The SMTP server host. Default localhost.

EMAIL\_PORT: The SMTP port Default 25.

EMAIL\_HOST\_USER: Username for the SMTP server.

EMAIL\_HOST\_PASSWORD: Password for the SMTP server.

EMAIL\_USE\_TLS: Whether to use a TLS secure connection.

EMAIL\_USE\_SSL: Whether to use an implicit TLS secure connection.

Nếu không có local SMTP, có thể sử dụng SMTP Server của Google. Lúc đó cấu hình như sau:

EMAIL\_HOST = 'smtp.gmail.com'

EMAIL\_HOST\_USER = 'your\_account@gmail.com'

EMAIL\_HOST\_PASSWORD = 'your\_password'

EMAIL\_PORT = 587

EMAIL\_USE\_TLS = True

Chạy shell của Python để test việc gởi mail:

python manage.py shell

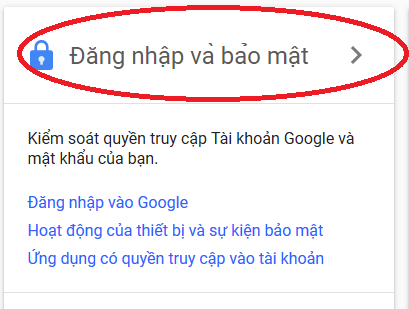
Sau đó nhập vào:

>>> from django.core.mail import send\_mail

>>> send\_mail('Django mail', 'This e-mail was sent with Django.', 'your\_account@gmail.com', ['your\_account@gmail.com'], fail\_ silently=False)

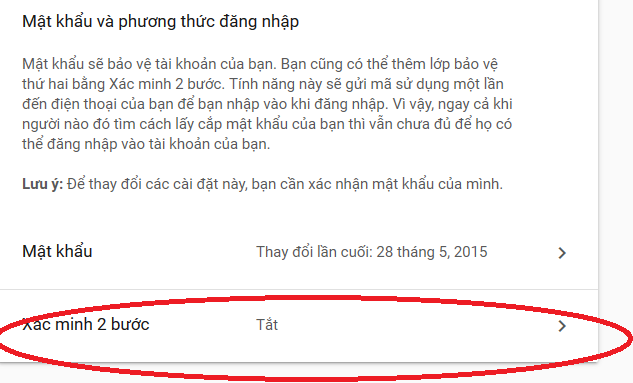
send\_mail() takes the subject, message, sender, and list of recipients as required arguments. By setting the optional argument **fail\_silently=False**, we are telling it to raise an exception if the e-mail couldn't be sent correctly. **If the output you see is 1, then your e-mail was successfully sent.**

Cấu hình Google Account để có thể gởi mail mà không cần vào Gmail:

vào link: [https://myaccount.google.com](https://myaccount.google.com)

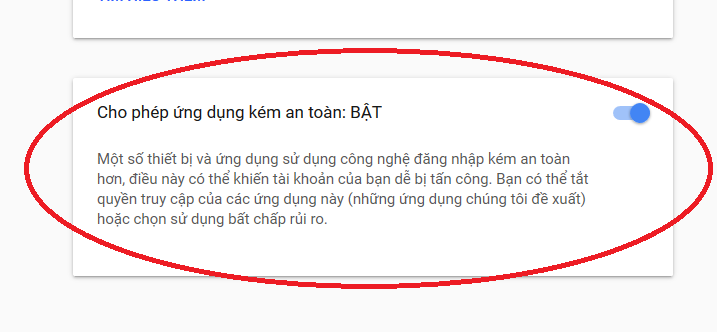
Sau khi chọn sẽ vào trang: https://myaccount.google.com/security

Tiếp theo:



Tiếp tục kéo xuống dưới:

Chuyển “Tắt” => “Bật” như hình



# Quan hệ giữa các model trong Django

## Quan hệ 1-n

Đây là quan hệ phổ biến nhất.

class Post(models.Model):

STATUS\_CHOICES = (

('draft', 'Draft'),

('published', 'Published')

)

objects = models.Manager() # The default manager.

published = PublishedManager() # Our custom manager.

title = models.CharField(max\_length=250)

slug = models.SlugField(max\_length=250, unique\_for\_date='publish')

#name of the reverse relationship, from User to Post, is specified in related\_name atribute

**author = models.ForeignKey(User, related\_name='blog\_posts')**

body = models.TextField()

publish = models.DateTimeField(default=timezone.now)

# the date will be saved automatically when creating an object

created = models.DateTimeField(auto\_now\_add=True)

# the date will be updated automatically when saving an object

updated = models.DateTimeField(auto\_now=True)

status = models.CharField(max\_length=10, choices = STATUS\_CHOICES, default='draft')

class Comment(models.Model):

**post = models.ForeignKey(Post, related\_name='comments')**

name = models.CharField(max\_length=80)

email = models.EmailField()

body = models.TextField()

created = models.DateTimeField(auto\_now\_add=True)

updated = models.DateTimeField(auto\_now=True)

active = models.BooleanField(default=True)

class Meta:

ordering = ('created',)

def \_\_str\_\_(self):

return 'Comment by {} on {}'.format(self.name, self.post)

Ở ví dụ trên, Post sẽ quan hệ với User thông qua thuộc tính khóa ngoại author (Post - User: n - 1, Comment sẽ quan hệ với Post qua thuộc tính khóa ngoại post (Post - Comment: 1 - n).

Thuộc tính related\_name cho phép chúng ta truy suất từ bên phía 1, thuộc tính khóa ngoại cho phép truy suất từ bên nhiều.

Với ví dụ trên thì từ Comment truy suất Post chỉ cần: comment.post, tương tự Post truy suất User thì post.author.

Còn từ bên 1 truy suất: từ User tìm các post: user.blog\_posts, từ Post tìm các comment: post.comments. Với blog\_posts và comments được chỉ ra trong related\_name.

Nếu không định nghĩa trong thuộc tính ralated\_name, Django mặc định sẽ sử dụng tên model + “\_set” (ví dụ: comment\_set) để lấy các đối tượng bên nhiều.

## Quan hệ 1-1

(cập nhật sau)

## Quan hệ n-n

(cập nhật sau)

# Sử dụng thư viện bên thứ 3

Ngoài viết app của bạn, đôi khi cần sử dụng thư viện bên thứ 3. Lúc này ta cần cài vào django. Khi cài cần lưu ý một số vấn đề sau:

- Version Django đang sử dụng là gì?

Để kiểm tra version Django đang sử dụng mở shell Python với lệnh:

python manage.py shell

Nhập vào:

>>import django

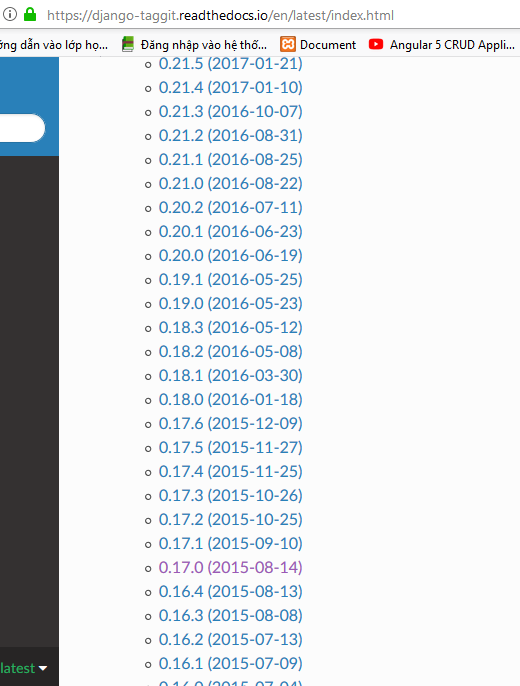
>>django.VERSION

- Dựa vào version Django để chọn lựa app cần tải (>= thời gian release của version Django hiện tại)

VD cần cài thư viện taggit (để tag cho post)

Trước tiên kiểm tra version Django => version 1.9.4 => thời gian release 5/5/2016

Thư viện taggit



Kiểm tra thấy thời gian để chọn version => chọn version taggit >= 0.18.2 => chọn 0.19.0

Cài đặt:

**pip install django-taggit==0.19.0**

## Sử dụng taggit cho Blog

- open the settings.py file of the mysite project and add taggit to your INSTALLED\_APPS setting like this:

INSTALLED\_APPS = (

# ...

'blog',

**'taggit',**

)

blog/model.py

**from taggit.managers import TaggableManager**

class Post(models.Model):

# ...

**tags = TaggableManager()**

Run the following command to create a migration for your model changes:

**python manage.py makemigrations blog**

You should get the following output:

Migrations for 'blog':

0003\_post\_tags.py:

- Add field tags to post

Now, run the following command to create the required database tables for django-taggit models and to synchronize your model changes:

**python manage.py migrate**

You will see an output indicating that migrations have been applied, as follows:

Applying taggit.0001\_initial... OK

Applying taggit.0002\_auto\_20150616\_2121... OK

Applying blog.0003\_post\_tags... OK

Test by shell

>>> from blog.models import Post

>>> post = Post.objects.get(id=1)

Then add some tags to it and retrieve its tags back to check that they were successfully added:

**>>> post.tags.add('music', 'jazz', 'django')**

**>>> post.tags.all()**

**[<Tag: jazz>, <Tag: django>, <Tag: music>]**

Finally, remove a tag and check the list of tags again:

>>> post.tags.remove('django')

>>> post.tags.all()

[<Tag: jazz>, <Tag: music>]

# Using the Django authentication framework

Django comes with a built-in authentication framework that can handle user authentication, sessions, permissions, and user groups. The authentication system includes views for common user actions such as login, logout, change password, and reset password.

The authentication framework is included in the default settings of your project. It consists of the django.contrib.auth application and the following two middleware classes found in the MIDDLEWARE\_CLASSES setting of your project:

1. AuthenticationMiddleware: Associates users with requests using sessions
2. SessionMiddleware: Handles the current session across requests

The authentication framework also includes the following models:

1. User: A user model with basic fields; the main fields of this model are: username, password, email, first\_name, last\_name, and is\_active.
2. Group: A group model to categorize users.
3. Permission: Flags to perform certain actions.

## Tạo login view

Cho phép người dùng login vào web (có thể không cần phải là admin và login ở frontend)

Các bước tiến hành:

- Lấy dữ liệu được post từ form

- Xác thực user với dữ liệu có trong CSDL

- Kiểm tra user được active hay chưa

- Cho user đăng nhập và bắt đầu phiên làm việc

## Using Django authentication views

Django provides the following views to deal with authentication:

1. login: Handles a log in form and logs in a user
2. logout: Logs out a user
3. logout\_then\_login: Logs out a user and redirects him to the log-in page

Django provides the following views to handle password changes:

1. password\_change: Handles a form to change user password
2. password\_change\_done: The success page shown to the user after changing his password

Django also includes the following views to allow users to reset their password:

1. password\_reset: Allows the user to reset his password. It generates a one-time use link with a token and sends it to the user's e-mail account.
2. password\_reset\_done: Shows the user that the e-mail to reset his password has been sent to his e-mail account.
3. password\_reset\_confirm: Lets the user set a new password.
4. password\_reset\_complete: The success page shown to the user after he resets their password.

## Log in and log out views

urls.py

from django.contrib.auth import views as auth\_views

from django.conf.urls import url

from . import views

urlpatterns = [

#view tu viet

#url(r'^login/$', views.user\_login, name='login')

#view cua django

url(r'^$', views.dashboard, name='dashboard'),

url(r'^login/$',

auth\_views.login,

name='login'),

url(r'^logout/$',

auth\_views.logout,

name='logout'),

url(r'^logout-then-login/$',

auth\_views.logout\_then\_login,

name='logout\_then\_login'),

]

Create a new directory inside the templates directory of your account application and name it registration. This is the default path where the Django authentication views expect your authentication templates to be.

registration/login.html

{% extends "base.html" %}

{% block title %}Log-in{% endblock %}

{% block content %}

<h1>Log-in</h1>

{% if form.errors %}

<p>

Your username and password didn't match.

Please try again.

</p>

{% else %}

<p>Please, use the following form to log-in:</p>

{% endif %}

<div class="login-form">

<form action="{% url 'login' %}" method="post">

{{ form.as\_p }}

{% csrf\_token %}

<input type="hidden" name="next" value="{{ next }}" />

<p><input type="submit" value="Log-in"></p>

</form>

</div>

{% endblock %}

Django uses the AuthenticationForm located at django.contrib.auth.forms by default.

This form tries to authenticate the user and raises a validation error if the login was not successful.

a hidden HTML <input> element to submit the value of a variable called next. This variable is first set by the log in view when you pass a next parameter in the request (for example. http://127.0.0.1:8000/account/login/?next=/account/).

The next parameter has to be a URL. If this parameter is given, the Django login view will redirect to the given URL after the user logs in.

registration/logged\_out.html

{% extends "base.html" %}

{% block title %}Logged out{% endblock %}

{% block content %}

<h1>Logged out</h1>

<p>You have been successfully logged out. You can <a href="{% url "login" %}">log-in again</a>.</p>

{% endblock %}

This is the template that Django will display after the user logs out.

Note that the logout\_then\_login view we included in our urlconf does not need any template since it performs a redirect to the log in view.

## Create a new view to display a dashboard

views.py

from django.contrib.auth.decorators import login\_required

@login\_required

def dashboard(request):

return render(request,

'account/dashboard.html',

{'section': 'dashboard'})

decorate our view with the login\_required decorator of the authentication framework

The login\_required decorator checks if the current user is authenticated. If the user is authenticated, it executes the decorated view; if the user is not authenticated, it redirects him to the login URL with the URL he was trying to access as a GET parameter named next. By doing so, the log in view redirects the user back to the URL he was trying to access after he is successfully logged in.

We also define a section variable. We are going to use this variable to track which section of the site the user is watching. Multiple views may correspond to the same section. This is a simple way to define which section each view corresponds to.

urls.py

urlpatterns = [

# ...

**url(r'^$', views.dashboard, name='dashboard'),**

]

settings.py (on bookmarks)

Edit the settings.py file of your project and add the following code to it:

from django.core.urlresolvers import reverse\_lazy

LOGIN\_REDIRECT\_URL = reverse\_lazy('dashboard')

LOGIN\_URL = reverse\_lazy('login')

LOGOUT\_URL = reverse\_lazy('logout')

These settings are:

1. LOGIN\_REDIRECT\_URL: Tells Django which URL to redirect after login if the contrib.auth.views.login view gets no next parameter
2. LOGIN\_URL: Is the URL to redirect the user to log in (e.g. using the login\_required decorator)
3. LOGOUT\_URL: Is the URL to redirect the user to log out

We are using reverse\_lazy() to build the URLs dynamically by their name. The reverse\_lazy() function reverses URLs just like reverse() does, but you can use it when you need to reverse URLs before your project's URL configuration is loaded.

## Thay đổi password user

password\_change->password\_change\_done

## Reset password user

Các bước:

reset\_form -> reset\_email->reset\_done->reset\_confirm->reset\_pass\_complete

Để test chức năng cần gởi email mà không cần phải sử dụng SMTP server để gởi, có thể in link ở console để test, cấu hình trong settings.py

EMAIL\_BACKEND = 'django.core.mail.backends.console.EmailBackend'

Ví dụ ở đây là gởi mail xác nhận thay vì người nhận sẽ nhận trong email rồi click vào link để xác nhận, ta có thể cho in link trong console và copy link dán trình duyệt để xử lý tiếp.

# Using the messages framework

Khi thao tác với form, người dùng cần nhận được thông báo như thành công, thất bại, cảnh báo… Django cung cấp **Using the messages framework để làm việc này. Framework này được cài sẵn khi tạo app, đó là** django.contrib.messages trong INSTALLED\_APPS sẽ thấy có sẵn.

Messages are stored in the database and displayed in the next request the user does.

Sử dụng như sau:

from django.contrib import messages

messages.error(request, 'Something went wrong')

You can create new messages using the add\_message() method or any of the following shortcut methods:

1. success(): Success messages to display after an action was successful
2. info(): Informational messages
3. warning(): Something has not yet failed but may fail imminently
4. error(): An action was not successful or something failed
5. debug(): Debug messages that will be removed or ignored in a production environment

Since the messages framework applies globally to the project, we can display messages for the user in our base template. Open the base.html template and add the following code between the <div> element with the id header and the <div> element with the id content:

{% if messages %}

<ul class="messages">

{% for message in messages %}

<li class="{{ message.tags }}">

{{ message|safe }}

<a href="#" class="close">✖</a>

</li>

{% endfor %}

</ul>

{% endif %}

# Study Case: Buid social web

Mục tiêu:

* Using the authentication framework
* Creating user registration views
* Extending the User model with a custom profile model
* Adding social authentication with python-social-auth

# Django nâng cao

## Sử dụng Generic relation foreign key

## Creating custom model fields

Ngoài các field do Django cung cấp, ta có thể tạo ra field tùy chọn cho model. Ví dụ sau tạo 1 field order kiểu là số và tự động tăng khi lưu vào CSDL nếu không được thiết lập, còn khi thiết lập thì sẽ lưu theo giá trị được thiết lập (na ná như ID tự tăng nhưng có thể thiết lập được - field này dùng để sắp xếp)

from django.db import models

from django.core.exceptions import ObjectDoesNotExist

class OrderField(models.PositiveIntegerField):

def \_\_init\_\_(self, for\_fields=None, \*args, \*\*kwargs):

self.for\_fields = for\_fields

super(OrderField, self).\_\_init\_\_(\*args, \*\*kwargs)

def pre\_save(self, model\_instance, add):

if getattr(model\_instance, self.attname) is None:

# no current value

try:

qs = self.model.objects.all()

if self.for\_fields:

# filter by objects with the same field values

# for the fields in "for\_fields"

query = {field: getattr(model\_instance, field) for

field in self.for\_fields}

qs = qs.filter(\*\*query)

# get the order of the last item

last\_item = qs.latest(self.attname)

value = last\_item.order + 1

except ObjectDoesNotExist:

value = 0

setattr(model\_instance, self.attname, value)

return value

else:

return super(OrderField,

self).pre\_save(model\_instance, add)

## Using class-based views

View có chức năng nhận yêu cầu, xử lý và phản hồi cho browser. Ngoài cách hiện thực view là một hàm thì ta có thể chuyển view thành class gọi là class-based view.

Lớp này sẽ kê thừa từ lớp View của Django. Sau đây là ví dụ chuyển view từ một phương thức thành một class.

def post\_list(request):

object\_list = Post.published.all()

paginator = Paginator(object\_list, 3) # 3 posts in each page

page = request.GET.get('page')

try:

posts = paginator.page(page)

except PageNotAnInteger:

# If page is not an integer deliver the first page

posts = paginator.page(1)

except EmptyPage:

# If page is out of range deliver last page of results

posts = paginator.page(paginator.num\_pages)

return render(request,

'blog/post/list.html',

{'page': page,

'posts': posts})

Lớp có dạng: TênModelViewcoso(Viewcoso), ví dụ như dưới đây tên model là Post và view cơ sở là Listview

class PostListView(ListView):

queryset = Post.published.all() #neu khong xac dinh thi queryset

#se la: Post.objects.all()

context\_object\_name = 'posts' #posts la ket qua truy van,

#neu ko chi ra thi mac dinh: object\_list

paginate\_by = 3 # so luong record tren 1 trang

template\_name = 'blog/post/list.html' #template su dung

Use a specific QuerySet instead of retrieving all objects. Instead of defining a queryset attribute, we could have specified model = Post and Django would have built the generic Post.objects.all() QuerySet for us.

• Use the context variable posts for the query results. The default variable is object\_list if we don't specify any context\_object\_name.

• Paginate the result displaying three objects per page.

• Use a custom template to render the page. If we don't set a default template, ListView will use blog/post\_list.html.

Khi sử dụng class-based view thì url sẽ thay đổi như sau:

urlpatterns = [

# post views

**# url(r'^$', views.post\_list, name='post\_list'),**

**url(r'^$', views.PostListView.as\_view(), name='post\_list'),**

…

]

In order to keep pagination working, we have to use the right page object that is passed to the template. Django's ListView passes the selected page in a variable called page\_obj, so you have to edit your post\_list.html template accordingly to include the paginator using the right variable, like this:

## {% include "pagination.html" with **page=page\_obj** %}

Biến được sử dụng để chỉ trang hiện tại là page\_obj.

Hàm view được trả về bởi as\_view sẽ là phần được sử dụng của mọi class-based View. Khi được gọi, công việc của nó là sử dụng dispatch để xử lý truy vấn từ người dùng và gọi đến các hàm xử lý tương ứng. Ví dụ khi truy vấn bằng GET thì sẽ gọi phương thức get, truy vấn POST sẽ gọi phương thức post, v.v...

## Sử dụng Mixins

In [object-oriented programming languages](bword://!!ARV6FUJ2JP,object-oriented%20programming%20language/), a **mixin** is a [class](bword://!!ARV6FUJ2JP,class%20%28computer%20science%29/) that contains methods for use by other classes without having to be the parent class of those other classes. How those other classes gain access to the mixin's methods depends on the language. Mixins are sometimes described as being "included" rather than "inherited".

Mixins là một dạng của đa thừa kế trong hướng đối tượng của Python. When you need to provide a specific behavior for several class-based views, it is recommended to use *mixins*.

from django.core.urlresolvers import reverse\_lazy

from django.views.generic.list import ListView

from django.views.generic.edit import CreateView, UpdateView, \

DeleteView

from .models import Course

class OwnerMixin(object):

def get\_queryset(self):

qs = super(OwnerMixin, self).get\_queryset()

return qs.filter(owner=self.request.user)

class OwnerEditMixin(object):

def form\_valid(self, form):

form.instance.owner = self.request.user

return super(OwnerEditMixin, self).form\_valid(form)

class OwnerCourseMixin(OwnerMixin):

model = Course

class OwnerCourseEditMixin(OwnerCourseMixin, OwnerEditMixin):

fields = ['subject', 'title', 'slug', 'overview']

success\_url = reverse\_lazy('manage\_course\_list')

template\_name = 'courses/manage/course/form.html'

class ManageCourseListView(OwnerCourseMixin, ListView):

template\_name = 'courses/manage/course/list.html'

class CourseCreateView(OwnerCourseEditMixin, CreateView):

pass

class CourseUpdateView(OwnerCourseEditMixin, UpdateView):

pass

class CourseDeleteView(OwnerCourseMixin, DeleteView):

template\_name = 'courses/manage/course/delete.html'

success\_url = reverse\_lazy('manage\_course\_list')

In this code, we create the OwnerMixin and OwnerEditMixin mixins. We will use these mixins together with the ListView, CreateView, UpdateView, and DeleteView views provided by Django. OwnerMixin implements the following method:

1. get\_queryset(): This method is used by the views to get the base QuerySet. Our mixin will override this method to filter objects by the owner attribute to retrieve objects that belong to the current user (request.user).

OwnerEditMixin implements the following method:

1. form\_valid(): This method is used by views that use Django's ModelFormMixin mixin, i.e., views with forms or modelforms such as CreateView and UpdateView. form\_valid() is executed when the submitted form is valid. The default behavior for this method is saving the instance (for modelforms) and redirecting the user to success\_url. We override this method to automatically set the current user in the owner attribute of the object being saved. By doing so, we set the owner for an object automatically when it is saved.

Our OwnerMixin class can be used for views that interact with any model that contains an owner attribute.

We also define an OwnerCourseMixin class that inherits OwnerMixin and provides the following attribute for child views:

1. model: The model used for QuerySets. Used by all views.

We define a OwnerCourseEditMixin mixin with the following attributes:

1. fields: The fields of the model to build the model form of the CreateView and UpdateView views.
2. success\_url: Used by CreateView and UpdateView to redirect the user after the form is successfully submitted. We use a URL with the name manage\_course\_list that we are going to create later.

Finally, we create the following views that subclass OwnerCourseMixin:

1. ManageCourseListView: Lists the courses created by the user. It inherits from OwnerCourseMixin and ListView.
2. CourseCreateView: Uses a modelform to create a new Course object. It uses the fields defined in OwnerCourseEditMixin to build a model form and also subclasses CreateView.
3. CourseUpdateView: Allows editing an existing Course object. It inherits from OwnerCourseEditMixin and UpdateView.
4. CourseDeleteView: Inherits from OwnerCourseMixin and the generic DeleteView. Defines success\_url to redirect the user after the object is deleted.

## Sử dụng Django access mixin

Từ Django 1.9 trở về sau cung cấp các lớp mixin cho xác thực trong contrib.auth giúp dễ dàng hơn trong việc điều khiển truy cập.

Hiện tại có 3 mixins được cung cấp là:

* -[LoginRequiredMixin](https://docs.djangoproject.com/en/1.9/topics/auth/default/#the-loginrequired-mixin) - restrict your view to only those users that are currently logged in.
* [UserPassesTestMixin](https://docs.djangoproject.com/en/1.9/topics/auth/default/#django.contrib.auth.mixins.UserPassesTestMixin) - restrict your view to only those users who pass a specified test.
* [PermissionRequiredMixin](https://docs.djangoproject.com/en/1.9/topics/auth/default/#the-permissionrequiredmixin-mixin) - restrict your view to only those users who have a specified permission or permissions.

Ví dụ cách sử dụng mixins

**from django.contrib.auth.mixins import PermissionRequiredMixin**

class MyView(**PermissionRequiredMixin**, View):

permission\_required = 'polls.can\_vote'

# Or multiple of permissions:

permission\_required = ('polls.can\_open', 'polls.can\_edit')

Ví dụ trên chỉ cho các user có quyền mới truy cập được lớp view trên.

from django.views.generic import ListView

# I am using Django 1.9, adjust the import path if you are using 1.10

**from django.contrib.auth.mixins import LoginRequiredMixin**

from django.core.urlresolvers import reverse\_lazy

class SampleListView(**LoginRequiredMixin**, ListView):

login\_url = reverse\_lazy('login')

redirect\_field\_name = 'redirect\_to'

context\_object\_name = 'sample\_list'

template\_name = 'sample\_list.html'

def dispatch(self, \*args, \*\*kwargs):

return super(SampleListView, self).dispatch(\*args, \*\*kwargs)

def get\_queryset(self):

# implementation omitted

User đăng nhập mới sử dụng được view trên.

## Sử dụng formset

Django comes with an abstraction layer to work with multiple forms on the same page. These groups of forms are known as formsets. Formsets manage multiple instances of certain Form or ModelForm. All forms are submitted at once and the formset takes care of things like the initial number of forms to display, limiting the maximum number of forms that can be submitted, and validating all forms.

Formsets include an is\_valid() method to validate all forms at once. You can also provide initial data for the forms and specify how many additional empty forms to display.

# Login và logout sử dụng view của Django

#### Configure the URL routes

First import the django.contrib.auth.views module and add a URL route for the login and logout views:

from django.conf.urls import url

from django.contrib import admin

from django.contrib.auth import views as auth\_views

from django.views.generic.base import TemplateView

urlpatterns = [

url(r'^$', TemplateView.as\_view(template\_name='registration/home.html'),

name='home'),

url(r'^login/$', auth\_views.login, name='login'),

url(r'^logout/$', auth\_views.logout, name='logout'),

url(r'^admin/', admin.site.urls),

]

#### Create a login template

By default, the django.contrib.auth.views.login view will try to render the registration/login.html template. So the basic configuration would be creating a folder named registration and place a login.html template inside.

Following a minimal login template:

{% extends 'base.html' %}

{% block title %}Login{% endblock %}

{% block content %}

<h2>Login</h2>

<form method="post">

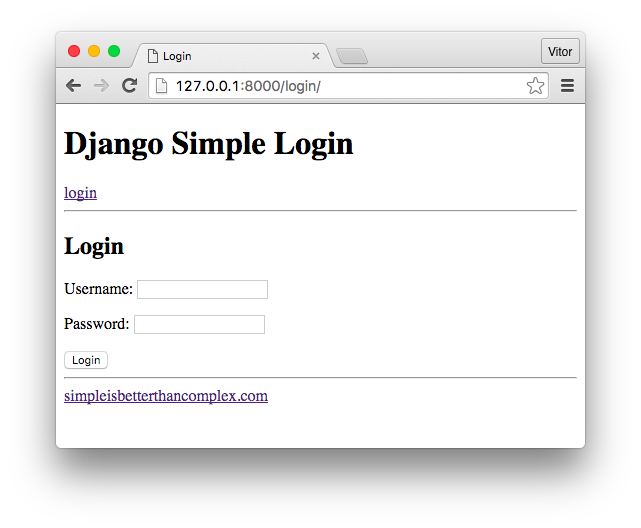
{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Login</button>

</form>

{% endblock %}



This simple example already validates username and password and authenticate correctly the user.

#### Customizing the login view

There are a few parameters you can pass to the login view to make it fit your project. For example, if you want to store your login template somewhere else than registration/login.html you can pass the template name as a parameter:

url(r'^login/$', auth\_views.login, {'template\_name': 'core/login.html'}, name='login'),

You can also pass a custom authentication form using the parameter authentication\_form, incase you have implemented a custom user model.

Now, a very important configuration is done in the settings.py file, which is the URL Django will redirect the user after a successful authentication.

Inside the settings.py file add:

LOGIN\_REDIRECT\_URL = 'home'

The value can be a hardcoded URL or a URL name. The default value for LOGIN\_REDIRECT\_URL is /accounts/profile/.

It is also important to note that Django will try to redirect the user to the next GET param.

Tham số next nếu cung cấp sẽ có dạng như sau:

<http://localhost:8000/admin/login/?next=/admin/>

Ví dụ gán tham số url vào link cho login

<p>You have been successfully logged out. You can <a href="{% url 'login' %}**?next=/course/mine/**">log-in again</a>.</p>

Lúc này khi login nếu thành công sẽ chuyển vào **/course/mine/**

#### Setting up logout view

After acessing the django.contrib.auth.views.logout view, Django will render the registration/logged\_out.html template. In a similar way as we did in the login view, you can pass a different template like so:

url(r'^logout/$', auth\_views.logout, {'template\_name': 'logged\_out.html'}, name='logout'),

Usually I prefer to use the next\_page parameter and redirect either to the homepage of my project or to the login page when it makes sense.

url(r'^logout/$', auth\_views.logout, {'next\_page': '/'}, name='logout'),

# Cách xử lý sau khi login dựa vào kiểu của User

- Có thể dùng tham số next như trên

- Cách nữa là chuyển đến 1 view nào đó ví dụ thiết lập LOGIN\_REDIRECT\_URL = 'home', và trong view Home này cho chuyển hướng đến url nào tùy thuộc vào kiểu của user.

Ví dụ trong views.py

from django.shortcuts import redirect, render

def home(request):

if request.user.is\_authenticated(): #kiem tra phien ban django

# co the la function, co the la attribute

if request.user.is\_teacher:

return redirect(quiz\_change\_list')

else:

return redirect(quiz\_list')

return render(request, 'classroom/home.html')

Ở đây nếu user là teacher -> chuyển đến quiz\_change\_list, ngược lại thì chuyển đến quiz\_list

# How to Implement Multiple User Types with Django

This is a very common problem many developers face in the early stages of the development of a new project, and it’s also a question I get asked a lot. So, I thought about sharing my experience with previous Django projects on how to handle multiple user types.

I’ve tried many different strategies. In this tutorial I will share my thoughts on this particular topic and share with you the strategies that worked best for me, depending on the requirements of the project.

Many of the design decisions depends on the requirements and business model of the application you are developing. I will try to cover as many different scenarios as possible. Read carefully and pick the best option for you.

#### Rules of Thumb

What you are going to read next is not written in a stone. It’s just some general recommendations that fits most cases. If you have a good reason, or if not following those recommendations will result in a better application design, go ahead and break the “rules”!

##### 1. No matter what strategy you pick, or what is your business model, always use one, and only one Django model to handle the authentication.

You can still have multiple user types, but generally speaking it’s a bad idea to store authentication information across multiple models/tables. Treat this model as an account rather than a user. Meaning, all users need an account to log in.

The business logic will be implemented in a different way, so no need to have username/password spread across multiple tables. After all, all accounts should share many common resources such as login, logout, password resets, password change.

##### 2. Never user the built-in Django User model directly, even if the built-in Django User implementation fulfill all the requirements of your application.

At least extend the AbstractUser model and switch the AUTH\_USER\_MODEL on your settings.

Requirements always change. You may need to customize the User model in the future, and switching the AUTH\_USER\_MODEL after your application is in production will be very painful. Mainly because you will need to update all the foreign keys to the User model. It can be done, but this simple measure (which, honestly, is effortless in the beginning of the project) can save you from headaches in the future.

This will give you freedom to add custom methods to the User model without having to rely on a OneToOne model or having to implement an Abstract model.

Also, the built-in Django model have some old design decisions (which are kept that way because of backwards compatibility) that are not compatible with many application requirements such as a nullable email field, the email field is not unique, the username field is case sensitive, which means you can have a user with username **ana** and another one with **Ana**, first name and last name fields which are not “internationally friendly” (some applications are better off having “full name” and “screen name” for example).

#### Strategies

How you are going to implement multiple user types depends on the requirements of your application. Below, a few questions you have to ask yourself:

* How much information specific to each type of user you need to maintain?
* Can the users have more than one role in the application? E.g. can a User be a Student and Teacher at the same time?
* How many different types of users the application will need to manage?

A very common case is to have a regular user and an admin user. For that case, you can use the built-in is\_staff flag to differentiate normal users from admin users. Actually the built-in User model has two similar fields, is\_staff and is\_superuser. Those flags are used in the Django Admin app, the is\_staff flag designates if the user can log in the Django Admin pages. Now, what this user can or cannot do, is defined by the permissions framework (where you can add specific permissions to a given user, e.g. can create/update users but cannot delete users). The is\_superuser flag is an additional flag to assign all permissions without having to add one by one. So here as you can see, the permissions are managed at two different levels.

If you need to maintain extra information related to the users, you have to ask yourself if this particular information is relevant to all users or if it is relevant to only some type of users. For example, a “student number” may only be relevant to Student users. In such cases, you are better off adding a profile model via one-to-one relationship. Now, if the extra info is relevant to all users (e.g., avatar image), the best thing to do is add an extra field directly to the User model.

If the users on your application can assume multiple roles at the same time (e.g. be a Student and Teacher), or your application will have only a few user types, you can control that information in the central User model and create flags like is\_student and is\_teacher:

class User(AbstractUser):

is\_student = models.BooleanField('student status', default=False)

is\_teacher = models.BooleanField('teacher status', default=False)

This is perhaps the easiest way to handle multiple user types.

Another option is, if the users can assume only one role, you could have a choices field like the example below:

class User(AbstractUser):

USER\_TYPE\_CHOICES = (

(1, 'student'),

(2, 'teacher'),

(3, 'secretary'),

(4, 'supervisor'),

(5, 'admin'),

)

user\_type = models.PositiveSmallIntegerField(choices=USER\_TYPE\_CHOICES)

This way you have a central point to check what is the type of the user. Usually using boolean flags works better!

If your application handle many user types, and users can assume multiple roles, an option is to create an extra table and create a many to many relationship:

class Role(models.Model):

'''

The Role entries are managed by the system,

automatically created via a Django data migration.

'''

STUDENT = 1

TEACHER = 2

SECRETARY = 3

SUPERVISOR = 4

ADMIN = 5

ROLE\_CHOICES = (

(STUDENT, 'student'),

(TEACHER, 'teacher'),

(SECRETARY, 'secretary'),

(SUPERVISOR, 'supervisor'),

(ADMIN, 'admin'),

)

id = models.PositiveSmallIntegerField(choices=ROLE\_CHOICES, primary\_key=True)

def \_\_str\_\_(self):

return self.get\_id\_display()

class User(AbstractUser):

roles = models.ManyToManyField(Role)

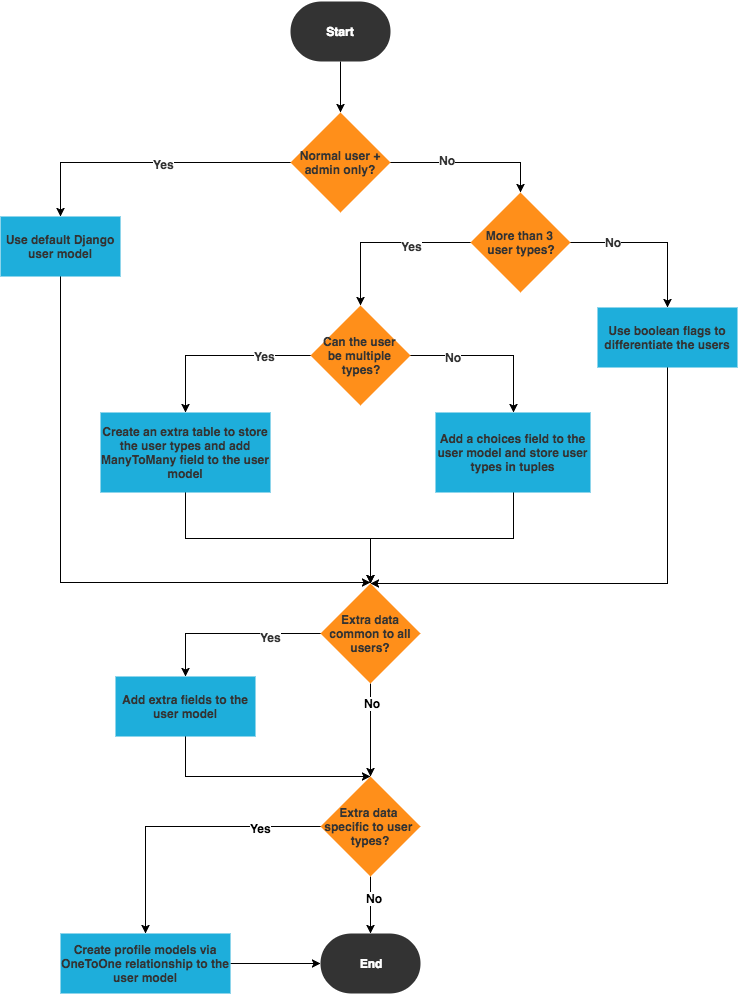
It wouldn’t make much sense to create roles programatically because they are very tied to the business logic. If you need a flexible permission management, it’s a better idea to use Django’s permission framework, where you can create groups and define specific permissions.

In this case I created some constants inside the Role model, so you can define behavior in the application using those constant values like if role == Role.ADMIN:.

It’s worth mentioning that this strategy is not very common. Evaluate first if creating custom permission groups wouldn’t be better, because Django already provides several tolling for this kind of thing.

#### Picking the Right Strategy

I created the flowchart below to help you decide what’s the right strategy for you. The orange diamonds are decision points; questions that you need to answer so to define the most suitable option. The blue rectangles are actions you will need to perform based on your decisions.



This flowchart is also a nice tool to remind you of what is important to take into account when designing your application.

#### Authentication and Authorization

Managing multiple user types or roles in a Web application boils down to permission management. Boolean flags like is\_student or is\_staff usually work as a fixed and pre-defined permission rules, that are incorporated in the source code of your application.

Before we jump into the implementation, it is important to highlight those two concepts:

* Authentication (Login)
* Authorization (Permission)

**Authentication** is the process of verifying if the person is who they claims to be. In practical terms, it’s the process of verifying username and password (login).

**Authorization** is the process of verifying what this particular person is allowed to do in the application. This can be achieved using view decorators, Django’s built-in permissions framework, or some third-party app like [django-guardian](https://github.com/django-guardian/django-guardian) or [django-rules](https://github.com/dfunckt/django-rules).

Handling authentication with Django is fairly simple, as it’s more standardized. Also, regardless of what the user can do in the application, the process of verifying their credentials should be pretty much the same, right? That’s usually the case in most Django applications. The thing is, most Django authentication forms encompasses aspects related to both authentication and authorization. That being said, authorization always comes after authentication.

Depending on your application needs, just one authentication form and view might be enough for all user types. But in some cases you will still need to implement different forms to authorize certain users to access certain modules or pages of your application.

Now, handling the authorization, i.e., what the users can or cannot do is the main challenge of implementing multiple user types. It’s a little bit more complicated because this can be done at different levels:

* It can be done at module or app level like Django Admin; only staff members can access the pages under /admin/.
* Within the same module or app, only a certain type of users can see and interact with some specific pages.
* Permission management within the same module where some users can only do certain actions (such as create or update objects) where other users can also delete objects.
* There’s also object-level permissions, where only the user who created it can interact with it. For example on Facebook, only you can edit your own posts (probably some super user / staff member can also interact with it, but from the business logic point of view, posts belongs to the users who posted it).

In the next section we are going to explore some real use cases. You will also find the complete source code so you can explore further or even use the code as the starting point for your implementation.

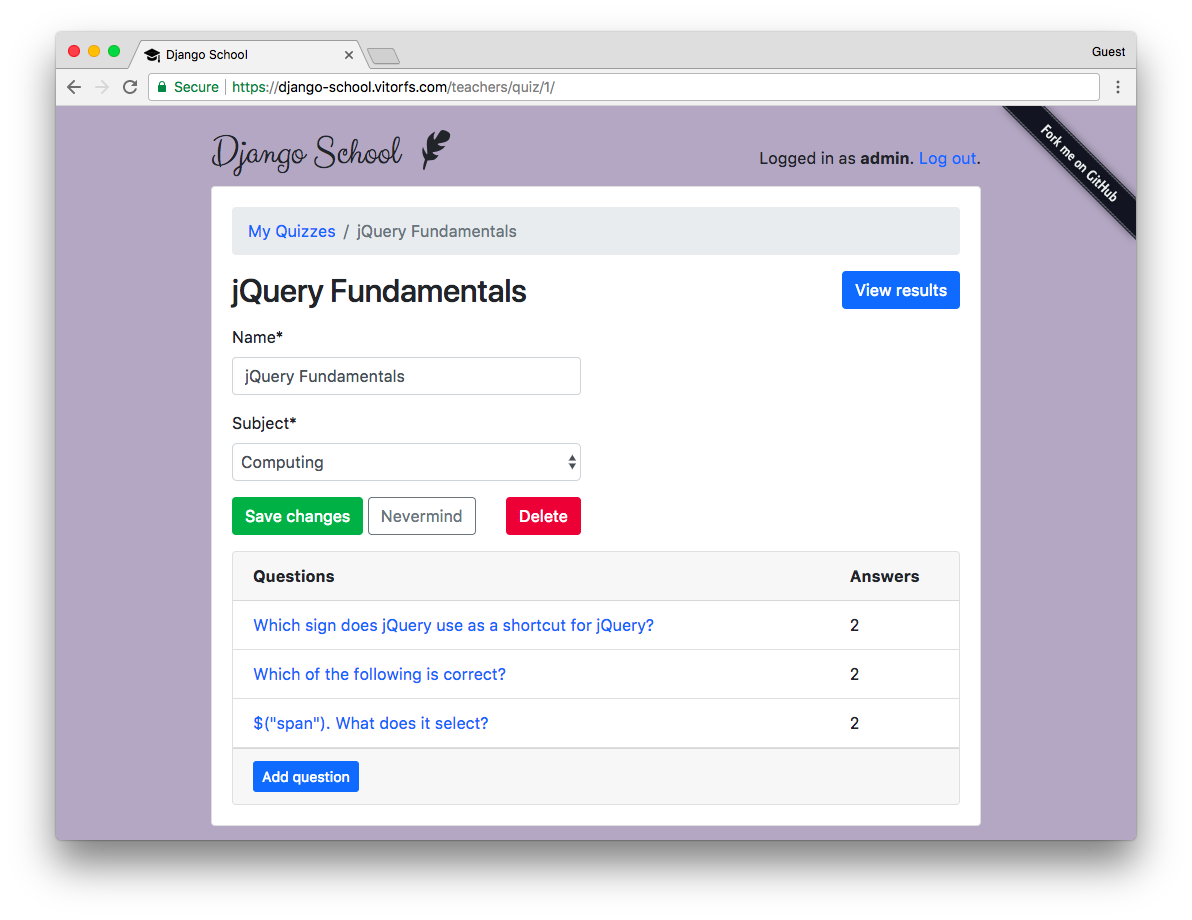
#### Practical Example

I was planning to create a simple example just to illustrate one case, but I ended up enjoying way too much the implementation. So I created something more elaborated (which I might use in future examples). The good thing is that now you have a working example that you can study or re-use some parts of the implementation.

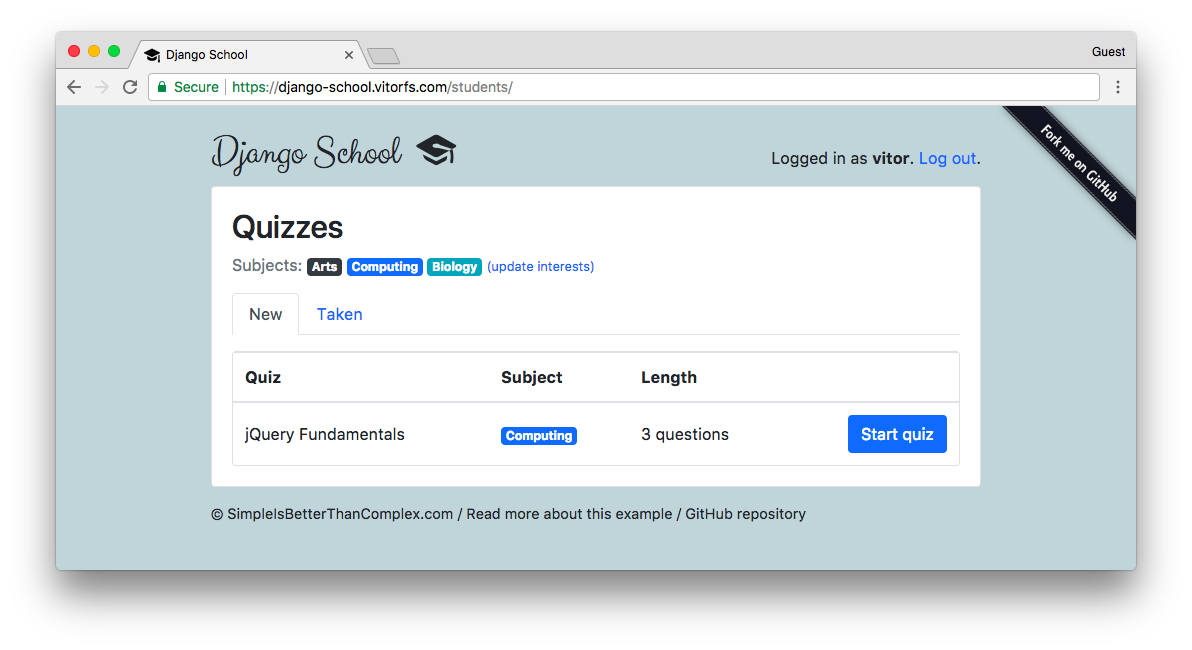
Anyway, this example is using Python 3.6, Django 2.0 and Bootstrap 4.

I named the project “Django School”, and it’s an application where teachers can create quizzes and students can sign up and select their interests. The application will show the students quizzes related to their interests.

So, in short: teachers create quiz and view results of the taken quizzes.



Students can answer to quizzes.



You can explore the views live using this URL: [django-school.vitorfs.com](https://django-school.vitorfs.com).

##### Models

Here is how I represented the models related to the user types:

**models.py**

from django.contrib.auth.models import AbstractUser

from django.db import models

class User(AbstractUser):

is\_student = models.BooleanField(default=False)

is\_teacher = models.BooleanField(default=False)

class Student(models.Model):

user = models.OneToOneField(User, on\_delete=models.CASCADE, primary\_key=True)

quizzes = models.ManyToManyField(Quiz, through='TakenQuiz')

interests = models.ManyToManyField(Subject, related\_name='interested\_students')

Note that I didn’t create a model named Teacher. For now, there is no need for that. Because the Student class for example, is just used to store information related to students.

The way we differentiate the users, and to know if a given user is a student or not, is by checking the is\_student flag.

For now you can ignore the quizzes and interests fields. Those are information related to the business logic of the application.

##### Sign Up / User Creation

In this case, I created one sign up view and form for each case.

**urls.py**

from django.conf.urls import url

from .views import HomeView,SignUpView, StudentSignUpView, TeacherSignUpView

urlpatterns = [

# url(r'^admin/', admin.site.urls),

url(r'^$', HomeView.as\_view(), name='home'),

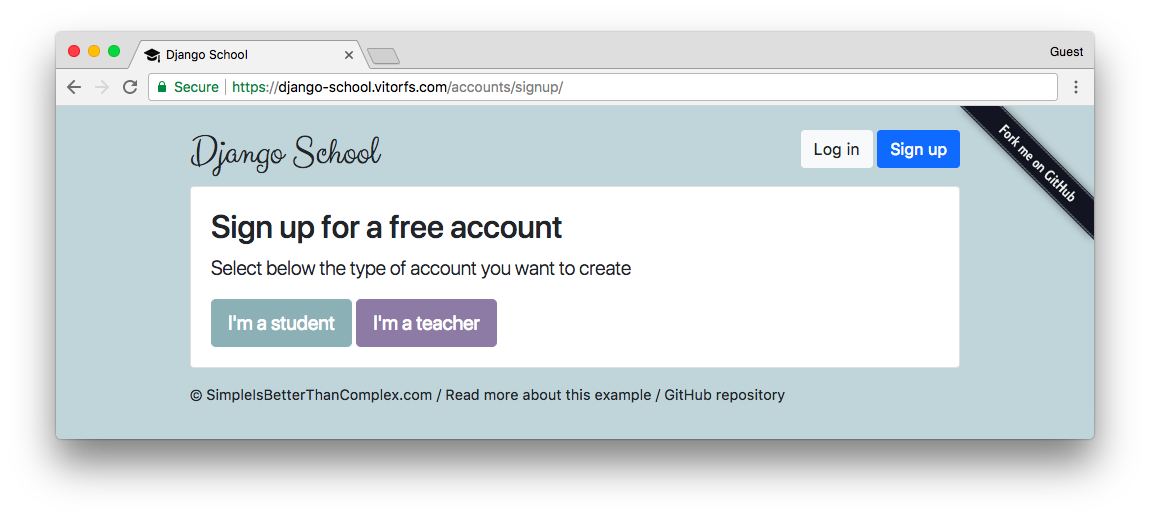
url(r'^signup/$', SignUpView.as\_view(), name='signup'),

url(r'^signup/student/$', StudentSignUpView.as\_view(), name='student\_signup'),

url(r'^signup/teacher/$', TeacherSignUpView.as\_view(), name='teacher\_signup'),

]

In the **urls.py** module you can see I have three sign up urls. The first one, asks the user if they want to sign up as a student or as a teacher:



Those buttons are just links that take the user either to the StudentSignUpView or to the TeacherSignUpView.

##### Student Sign Up

Below, just a basic view processing using the generic class-based view CreateView and the most important piece: the StudentSignUpForm.

**views/students.py**

from django.contrib.auth import login

from django.shortcuts import redirect

from django.views.generic import CreateView

from .forms import StudentSignUpForm

from .models import User

class StudentSignUpView(CreateView):

model = User

form\_class = StudentSignupForm

success\_url = reverse\_lazy('home')

template\_name = 'registration/signup\_form.html'

# du lieu co duoc truoc khi goi ra template them

# du lieu lien quan den user\_type => student

def get\_context\_data(self, \*\*kwargs):

context = super(StudentSignUpView, self).get\_context\_data(\*\*kwargs)

context['user\_type'] = 'student'

return context

def form\_valid(self, form):

result = super(TeacherSignUpView, self).form\_valid(form)

cd = form.cleaned\_data

#xac thuc truoc khi goi login

user = authenticate(username = cd['username'],

password = cd['password1'])

# user = form.save()

login(self.request, user)

return result

**forms.py**

from django import forms

from django.contrib.auth.forms import UserCreationForm

from django.db import transaction

from classroom.models import Student, Subject, User

class StudentSignUpForm(UserCreationForm):

interests = forms.ModelMultipleChoiceField(

queryset=Subject.objects.all(),

widget=forms.CheckboxSelectMultiple,

required=True

)

class Meta(UserCreationForm.Meta):

model = User

@transaction.atomic

def save(self):

user = super().save(commit=False)

user.is\_student = True

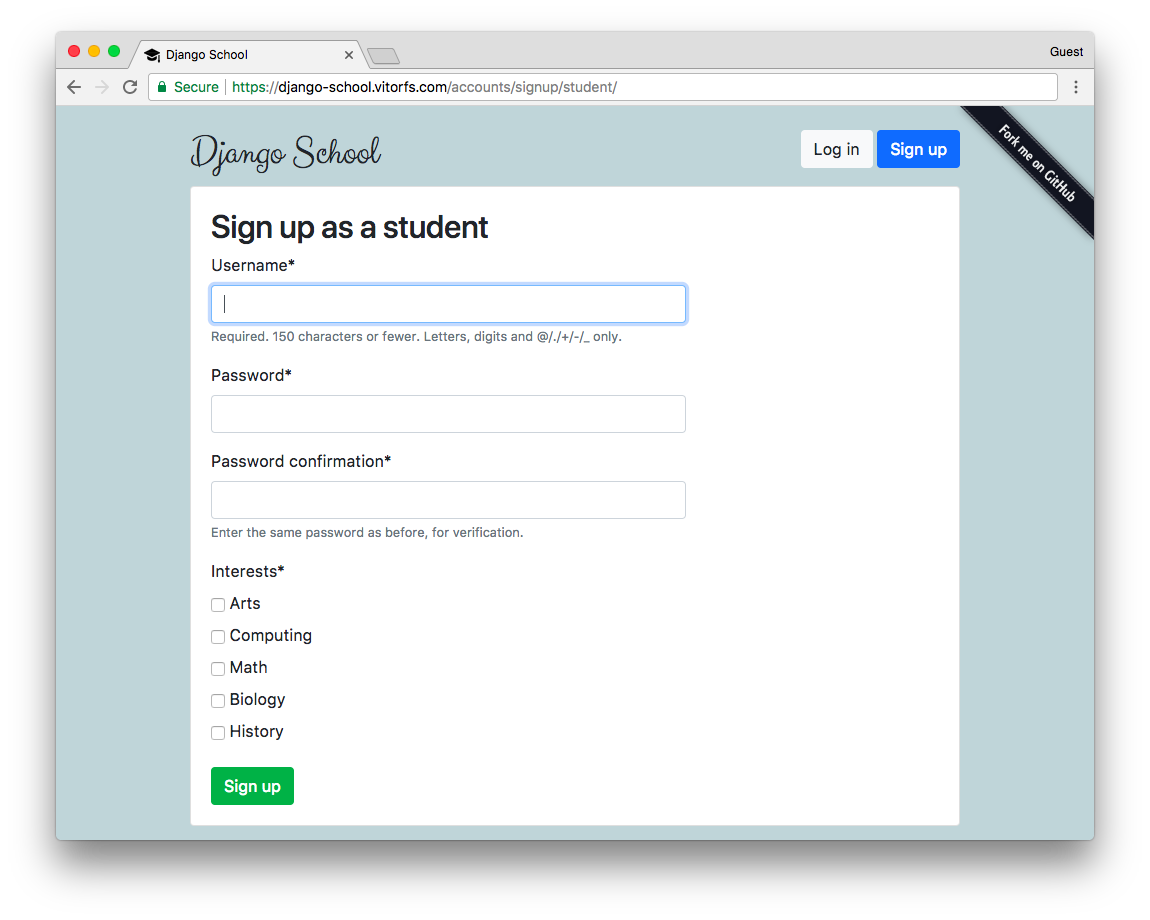
user.save()

student = Student.objects.create(user=user)

student.interests.add(\*self.cleaned\_data.get('interests'))

return user

As the base I used the built-in UserCreationForm, which defines the username and password fields. Inside the save method, I’m setting the is\_student flag to True. And then after that I create a Student profile to store the extra information. Finally, the student interests ad saved into the profile model. Note that the save method is decorated with the transaction.atomic, to make sure those three operations are done in a single database transaction and avoid data inconsistencies in case of error.



##### Teacher Sign Up

The teacher sign up is very similar but more simple:

**views/teachers.py**

from django.contrib.auth import login

from django.shortcuts import redirect

from django.views.generic import CreateView

class TeacherSignUpView(CreateView):

model = User #model tuong tac

form\_class = TeacherSignupForm #form su dung cho view

success\_url = reverse\_lazy('home') #sau khi xu ly thanh cong

template\_name = 'registration/signup\_form.html' #template su dung

def get\_context\_data(self, \*\*kwargs): #dua bien: user\_type ra template

context = super(TeacherSignUpView, self).get\_context\_data(\*\*kwargs)

context['user\_type'] = 'teacher'

return context

def form\_valid(self, form):

result = super(TeacherSignUpView, self).form\_valid(form)

cd = form.cleaned\_data

#xac thuc truoc khi goi login

user = authenticate(username = cd['username'],

password = cd['password1'])

# user = form.save()

login(self.request, user)

return result

**forms.py**

from django.contrib.auth.forms import UserCreationForm

from classroom.models import User

class TeacherSignUpForm(UserCreationForm):

class Meta(UserCreationForm.Meta):

model = User

def save(self, commit=True):

user = super().save(commit=False)

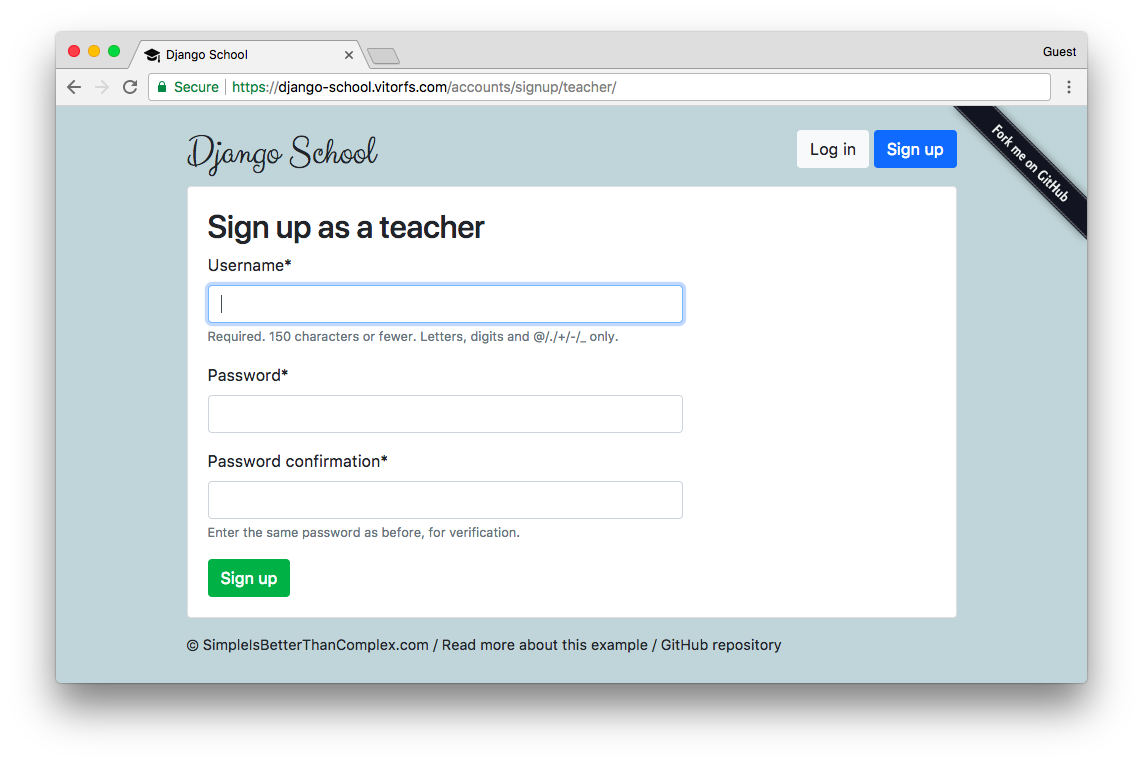
user.is\_teacher = True

if commit:

user.save()

return user

Here is just a matter of switching the is\_teacher flag to True. The template looks like this:



##### Protecting the Views

A simple way to protect your views is to create view decorators and apply them to specific views.

In the examples below I used the built-in user\_passes\_test to create two custom decorators. One to test if the user is a student and the other one to test if the user is a teacher.

**decorators.py**

from django.contrib.auth import REDIRECT\_FIELD\_NAME

from django.contrib.auth.decorators import user\_passes\_test

def student\_required(function=None, redirect\_field\_name=REDIRECT\_FIELD\_NAME, login\_url='login'):

'''

Decorator for views that checks that the logged in user is a student,

redirects to the log-in page if necessary.

'''

actual\_decorator = user\_passes\_test(

lambda u: u.is\_active and u.is\_student,

login\_url=login\_url,

redirect\_field\_name=redirect\_field\_name

)

if function:

return actual\_decorator(function)

return actual\_decorator

def teacher\_required(function=None, redirect\_field\_name=REDIRECT\_FIELD\_NAME, login\_url='login'):

'''

Decorator for views that checks that the logged in user is a teacher,

redirects to the log-in page if necessary.

'''

actual\_decorator = user\_passes\_test(

lambda u: u.is\_active and u.is\_teacher,

login\_url=login\_url,

redirect\_field\_name=redirect\_field\_name

)

if function:

return actual\_decorator(function)

return actual\_decorator

Basically those functions student\_required and teacher\_required are pieces of code that will be executed every time the view is executed.

If we want a view to be accessible only students, here is how we should use it:

**views/students.py**

from django.contrib.auth.decorators import login\_required

from django.shortcuts import get\_object\_or\_404, render

from .decorators import student\_required

from .models import Quiz

@login\_required

@student\_required # <-- here!

def take\_quiz(request, pk):

quiz = get\_object\_or\_404(Quiz, pk=pk)

student = request.user.student

# body of the view...

return render(request, 'classroom/students/take\_quiz\_form.html', {

'quiz': quiz,

'question': question,

'form': form,

'progress': progress

})

The example above is showing the usage of two decorators in a function based view. In this example the take\_quiz view is protected against users that are not logged in and from users who are not students.

Now if you are using class-based views, here is how we use it:

**views/students.py**

from django.utils.decorators import method\_decorator

from django.contrib.auth.decorators import login\_required

from ..decorators import student\_required

@method\_decorator([login\_required, student\_required], name='dispatch')

class StudentInterestsView(UpdateView):

model = Student

form\_class = StudentInterestsForm

template\_name = 'classroom/students/interests\_form.html'

success\_url = reverse\_lazy('students:quiz\_list')

def get\_object(self):

return self.request.user.student

def form\_valid(self, form):

messages.success(self.request, 'Interests updated with success!')

return super().form\_valid(form)

We need this extra import method\_decorator. Then we pass a list of view decorators. It’s pretty much the same thing.

What happens here is, if the user is not authorized, they are taken to the login page.

##### Verifying User Type

To check the type of the user to add a custom behavior in a template or anywhere else it’s just a matter of performing a simple if.

For example, in the two first screen shots you can see that the background color of the student area is green, and the background color for the teacher is purple. To do that, I have two sets of CSS, then I apply them to the HTML template dynamically:

**templates/base.html**

<head>

{% if user.is\_authenticated and user.is\_teacher %}

<link rel="stylesheet" type="text/css" href="{% static 'css/teachers.css' %}">

{% else %}

<link rel="stylesheet" type="text/css" href="{% static 'css/students.css' %}">

{% endif %}

</head>

You can apply the same logic for views that both students and teachers can access, but they can see different resources. For example a menu is different, or teachers can see a specific button. Then you can build this logic in the template. Just don’t forget to protect the view itself using the decorators! Otherwise a use could forge the URL and access resources that they are not supposed to have access to.

#### Conclusions

In short, you will always have one User model to handle authentication. Do not spread username and passwords across multiple models. Usually extending the default User model and adding boolean flags such as is\_student and is\_staff work for most cases. Permissions can be managed at a higher level using view decorators. If you need flexible permission management (no clear roles between the users or many specific roles), you will need to use Django’s permissions framework or a third-party library.

I spent some time in this example to make it look good. If you are interested in learning more about Django, this project is a good opportunity to dive deeper. Particularly you can get some insights about how to write custom forms and formsets, how to work with the generic class based-views and how to use some Bootstrap 4 components to compose Django templates.

If you have any questions, feel free to ask in the comments below!

# Gặp lỗi khi gọi lệnh: python manage.py makemigrations

Lỗi như sau:

SystemCheckError: System check identified some issues:

ERRORS:

auth.User.groups: (fields.E304) Reverse accessor for 'User.groups' clashes with reverse accessor for 'User.groups'.

HINT: Add or change a related\_name argument to the definition for 'User.groups' or 'User.groups'.

auth.User.user\_permissions: (fields.E304) Reverse accessor for 'User.user\_permissions' clashes with reverse accessor for 'User.user\_permissions'.

HINT: Add or change a related\_name argument to the definition for 'User.user\_permissions' or 'User.user\_permissions'.

school.User.groups: (fields.E304) Reverse accessor for 'User.groups' clashes with reverse accessor for 'User.groups'.

HINT: Add or change a related\_name argument to the definition for 'User.groups' or 'User.groups'.

school.User.user\_permissions: (fields.E304) Reverse accessor for 'User.user\_permissions' clashes with reverse accessor for 'User.user\_permissions'.

HINT: Add or change a related\_name argument to the definition for 'User.user\_permissions' or 'User.user\_permissions'.

Nguyên nhân lỗi: khi thực hiện việc kế thừa và mở rộng từ AbstractUser, cần bật biến AUTH\_USER\_MODEL trong setting.

Khắc phục như sau:

AUTH\_USER\_MODEL = "APPNAME.User" => thay APPNAME bằng ứng dụng đang chạy.

# get\_context\_data()

Hàm này thêm dữ liệu trước khi gởi để hiện thị lên template. Ví dụ get\_queryset() trả về một listview, giờ cần thêm dữ liệu: như page\_title => ‘Authors’ vào listview này để hiển thị thì sử dụng hàm này.

This method is used to populate a dictionary to use as the template context. For example, ListViews will populate the result from get\_queryset() as author\_list in the above example. You will probably be overriding this method most often to add things to display in your templates.

def get\_context\_data(self, \*\*kwargs):

data = super().get\_context\_data(\*\*kwargs)

data['page\_title'] = 'Authors'

return data

And then in your template you can reference these variables.

<h1>{{ page\_title }}</h1>

<ul>

{% for author in author\_list %}

<li>{{ author.name }}</li>

{% endfor %}

</ul>

# Django CRUD (Create, Retrieve, Update, Delete)

One of the most common task when developing web application is to write create, read, update and delete functionality (CRUD) for each table you have.

In this post I briefly cover the step needed to create a CRUD app in Django, the steps we will need are:

* Install Django & start new project
* Create an App
* Create the Model
* Create the Admin Interface (optional)
* Create the View
* Define the URLs (i.e. URL to View mapping)
* Create the Templates

### Install Django and Start New Project

First we need to install Django and start new Django project, I’ll name it my\_proj:

pip install django

django-admin startproject my\_proj

cd my\_proj

### Create new App

From the Django project directory we will create the new app called “books” to store our books collection:

./manage.py startapp books

We will also need to register the new app in our Django project, add the app “books” to the INSTALLED\_APPS in your my\_proj/settings.py:

INSTALLED\_APPS = (

:

'books',

:

)

### Create the Model

The model file would be books/models.py:

from django.db import models

from django.urls import reverse

class Book(models.Model):

name = models.CharField(max\_length=200)

pages = models.IntegerField()

def \_\_str\_\_(self):

return self.name

def get\_absolute\_url(self):

return reverse('book\_edit', kwargs={'pk': self.pk})

After defining the model you need to provision it to the database:

./manage.py makemigrations

./manage.py migrate

To create the table for the new model.

### Admin Interface (optional)

Django will give you free CRUD interface from the admin site, just define the file books/admin.py as:

from django.contrib import admin

from books.models import Book

admin.site.register(Book)

### The Views

We will use Django Class-based views to create our app pages, the file books/views.py would look like:

from django.http import HttpResponse

from django.views.generic import ListView, DetailView

from django.views.generic.edit import CreateView, UpdateView, DeleteView

from django.urls import reverse\_lazy

from books.models import Book

class BookList(ListView):

model = Book

class BookView(DetailView):

model = Book

class BookCreate(CreateView):

model = Book

fields = ['name', 'pages']

success\_url = reverse\_lazy('book\_list')

class BookUpdate(UpdateView):

model = Book

fields = ['name', 'pages']

success\_url = reverse\_lazy('book\_list')

class BookDelete(DeleteView):

model = Book

success\_url = reverse\_lazy('book\_list')

### Define the URLs

We need to define app URLs in the file books/urls.py (create the file):

from django.urls import path

from . import views

urlpatterns = [

path('', views.BookList.as\_view(), name='book\_list'),

path('view/<int:pk>', views.BookView.as\_view(), name='book\_view'),

path('new', views.BookCreate.as\_view(), name='book\_new'),

path('view/<int:pk>', views.BookView.as\_view(), name='book\_view'),

path('edit/<int:pk>', views.BookUpdate.as\_view(), name='book\_edit'),

path('delete/<int:pk>', views.BookDelete.as\_view(), name='book\_delete'),

]

This URLs wouldn’t work unless you include the books/urls.py in the main URLs file my\_proj/urls.py:

# Make sure you import "include" function

from django.urls import include

urlpatterns = [

:

path('books/', include('books.urls')),

:

]

### Templates

books/templates/books/book\_list.html This file will be used by the ListView:

<h1>Books</h1>

<table border="1">

<thead>

<tr>

<th>Name</th>

<th>Pages</th>

<th>View</th>

<th>Edit</th>

<th>Delete</th>

</tr>

</thead>

<tbody>

{% for book in object\_list %}

<tr>

<td>{{ book.name }}</td>

<td>{{ book.pages }}</td>

<td><a href="{% url "book\_view" book.id %}">view</a></td>

<td><a href="{% url "book\_edit" book.id %}">edit</a></td>

<td><a href="{% url "book\_delete" book.id %}">delete</a></td>

</tr>

{% endfor %}

</tbody>

</table>

<a href="{% url "book\_new" %}">New</a>

books/templates/books/book\_detail.html This file will be used by detail views:

<h1>Book Details</h1>

<h2>Name: {{object.name}}</h2>

Pages: {{ object.pages }}

books/templates/books/book\_form.html This file will be used by Edit and Update views:

<h1>Book Edit</h1>

<form method="post">{% csrf\_token %}

{{ form.as\_p }}

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

</form>

books/templates/books/book\_confirm\_delete.html This file will be used by DeleteView:

<h1>Book Delete</h1>

<form method="post">{% csrf\_token %}

Are you sure you want to delete "{{ object }}" ?

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

</form>

### Test It

So everything in place, now we can run the development web server:

./manage.py runserver

Then access it through a web browser <http://localhost:8000/books/>

To test the admin interface we need to create a user first:

./manage.py createsuperuser

and access it <http://localhost:8000/admin/>

### Function Based View Version

The example above uses Class Based Views (or CBV for short) to implement the views, what I will cover now is how to implement the same functionality but with Function Based Views i.e. using functions instead of classes, we will be using the same templates:

books/views.py:

from django.shortcuts import render, redirect, get\_object\_or\_404

from django.forms import ModelForm

from books.models import Book

class BookForm(ModelForm):

class Meta:

model = Book

fields = ['name', 'pages']

def book\_list(request, template\_name='books/book\_list.html'):

book = Book.objects.all()

data = {}

data['object\_list'] = book

return render(request, template\_name, data)

def book\_view(request, pk, template\_name='books/book\_detail.html'):

book= get\_object\_or\_404(Book, pk=pk)

return render(request, template\_name, {'object':book})

def book\_create(request, template\_name='books/book\_form.html'):

form = BookForm(request.POST or None)

if form.is\_valid():

form.save()

return redirect('book\_list')

return render(request, template\_name, {'form':form})

def book\_update(request, pk, template\_name='books/book\_form.html'):

book= get\_object\_or\_404(Book, pk=pk)

form = BookForm(request.POST or None, instance=book)

if form.is\_valid():

form.save()

return redirect('book\_list')

return render(request, template\_name, {'form':form})

def book\_delete(request, pk, template\_name='books/book\_confirm\_delete.html'):

book= get\_object\_or\_404(Book, pk=pk)

if request.method=='POST':

book.delete()

return redirect('book\_list')

return render(request, template\_name, {'object':book})

books/urls.py:

from django.urls import path

from books import views

urlpatterns = [

path('', views.book\_list, name='book\_list'),

path('view/<int:pk>', views.book\_view, name='book\_view'),

path('new', views.book\_create, name='book\_new'),

path('edit/<int:pk>', views.book\_update, name='book\_edit'),

path('delete/<int:pk>', views.book\_delete, name='book\_delete'),

]

You clone the complete example from <https://github.com/rayed/django_crud>

p.s. personally I prefer using FBV over CBV, the article [Django’s CBVs were a mistake](http://lukeplant.me.uk/blog/posts/djangos-cbvs-were-a-mistake/) explains why.

**UPDATE:** You might be interested in [Django CRUD Parent/Child Edition](https://rayed.com/posts/2016/01/django-crud-parentchild-edition/)

# How do you query with a condition on a ManyToMany models in Django through intermediary model?

Here's how \*I\* made this work; perhaps it'll help you figure out what's going on in your specific case. My models look like:

1. from django.db import models
3. class Person(models.Model):
4. name = models.CharField(max\_length=200)
5. groups = models.ManyToManyField('Group', through='GroupMember', related\_name='people')
7. class Group(models.Model):
8. name = models.CharField(max\_length=200)
10. class GroupMember(models.Model):
11. person = models.ForeignKey(Person, related\_name='membership')
12. group = models.ForeignKey(Group, related\_name='membership')
13. type = models.CharField(max\_length=100)

And given that definition, queries like the following work just fine:

1. # Which groups is Jim in?
2. Group.objects.filter(people=jim)

5. # But which groups does Jim admin?
6. Group.objects.filter(people=jim, membership\_\_type='admin')

9. # And which groups is Jim just a member of?
10. Group.objects.filter(people=jim, membership\_\_type='member')

I've put the full code I used to check that this works at [https://gist.github.com](https://gist.github.com/827937;)