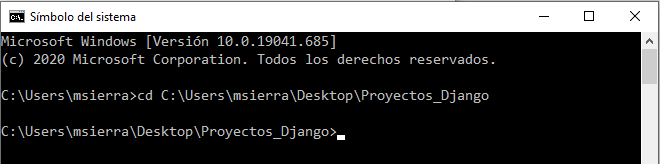
# Paso a paso para la instalación, configuración y creación de una aplicación Web de un Blog con Django y Python

# Pasos:

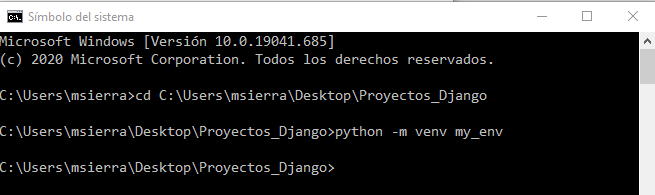
1. Crear un entorno Python aislado (environment)
2. Instalación de Django con pip
3. Crear un Proyecto en Django
   1. Completar la configuración del Proyecto y crear las tablas asociadas a los modelos
   2. Ejecutar el servidor de desarrollo
4. Crear una aplicación
   1. Diseñar el esquema de datos del blog
   2. Activando la aplicación
   3. Crear y aplicar migraciones
   4. Crear un usuario administrador
   5. Agregar modelos al sitio de administración
   6. Modificar la página de administración de Posts
   7. Creemos un Posts en memoria y luego en BBDD
   8. Obtener todos los posts
   9. Agregar un administrador personalizado
   10. Crear vistas de lista y detalle
5. Funciones Avanzadas
   1. Compartir publicaciones por Correo Electrónico
   2. Crear el modelo de comentarios
6. Como levantar la Página Web en desarrollo:

## Crear un entorno Python aislado

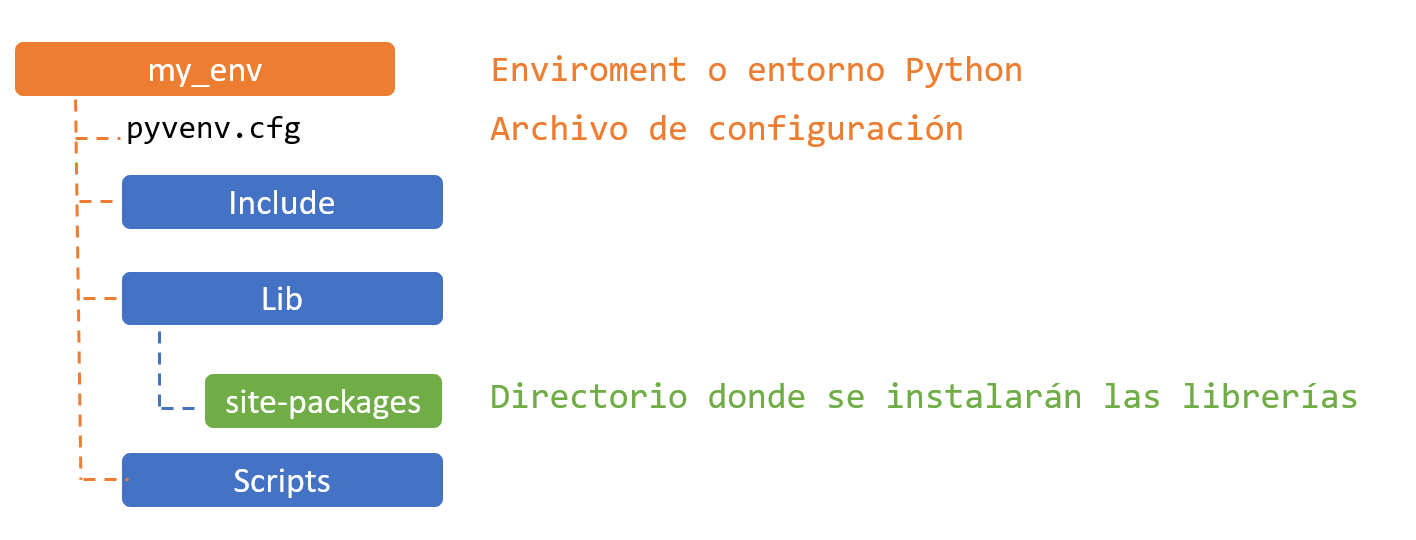
1. Ejecutamos la aplicación Símbolo del Sistema de Windows. Escribimos cmd desde la opción buscar de Windows.
2. Verificamos que tenemos instalado el virtualenv:
   1. Para instalarlo, podemos ejecutar desde el cmd el comando: pip install virtualenv
3. Actualicemos la versión del pip, podemos utilizar el comando: python -m pip install --upgrade pip
4. Nos ubicamos en el directorio donde queremos crear el **entorno virtual del proyecto**: En mi caso: cd C:\Users\user\Desktop\Proyectos\_Django



1. Para crear nuestro entorno virtual my\_env, ejecutamos el comando: python -m venv my\_env

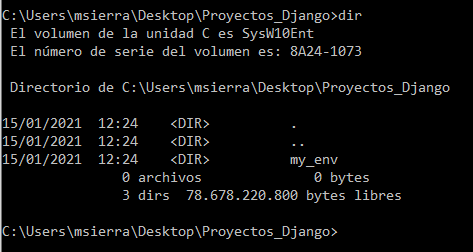


1. Verificamos que la estructura de carpetas de nuestro entorno virtual se ha creado:

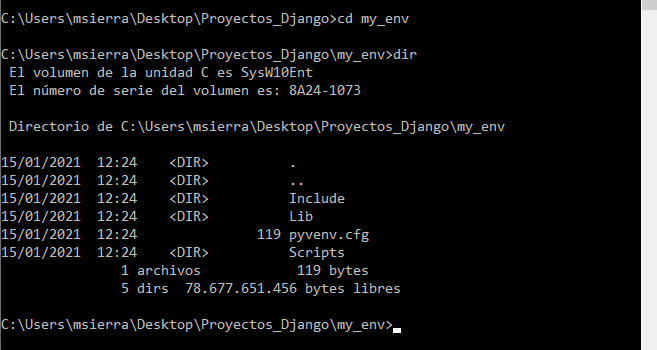


1. Utilizando el comando dir, podemos verificar la creación de la estructura:

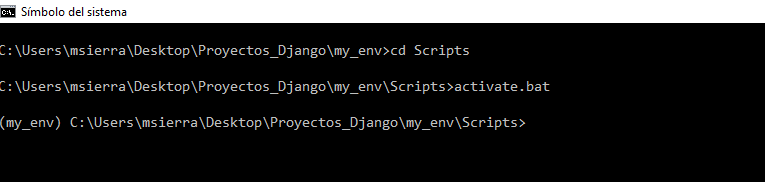
* C:\Users\user\Desktop\Proyectos\_Django> dir



* C:\Users\user\Desktop\Proyectos\_Django> cd my\_env
* C:\Users\user\Desktop\Proyectos\_Django\my\_env> dir



1. Activar el entorno virtual
   1. No posicionamos en el directorio principal, en nuestro caso con cd C:\Users\user\Desktop\Proyectos\_Django
   2. Ejecutamos el comando: source my\_env/scripts/activate. Si nos genera error, podemos ubicarnos en el directorio donde se encuentra el fichero activate.bat, en nuestro caso en: C:\Users\user\Desktop\Proyectos\_Django\my\_env\Scripts

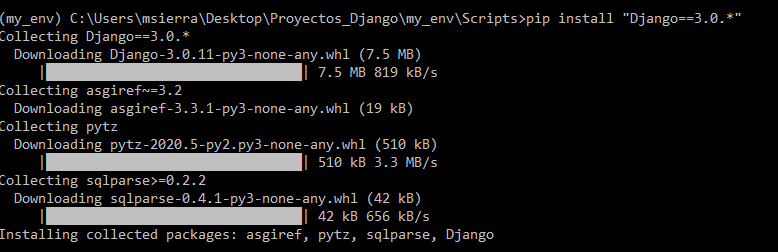


* 1. Para validar que el entorno esta activo, debemos observar que el prompt ha cambiado a:

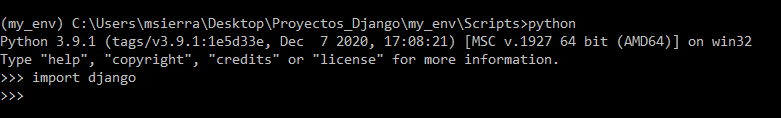
(my\_env) C:\Users\user\Desktop\Proyectos\_Django\my\_env\Scripts>

## Instalación de Django con pip

1. Desde el prompt del Shell ejecutamos pip install "Django==3.0.\*"



1. Para comprobar si Django se ha instalado correctamente. Ejecutamos Python desde la terminal e importamos la librería Django y verificamos su versión, de la siguiente manera:
   1. Desde la ventana **simbolo del sistema** que tenemos abierta ejecutamos: python
   2. Desde el prompt >> import django



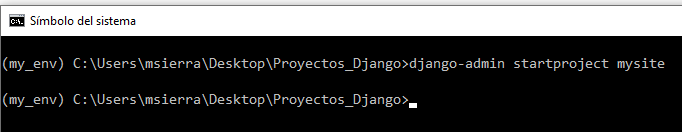
* 1. Verificamos la versión instalada, ejecutando: >> django.get\_version()



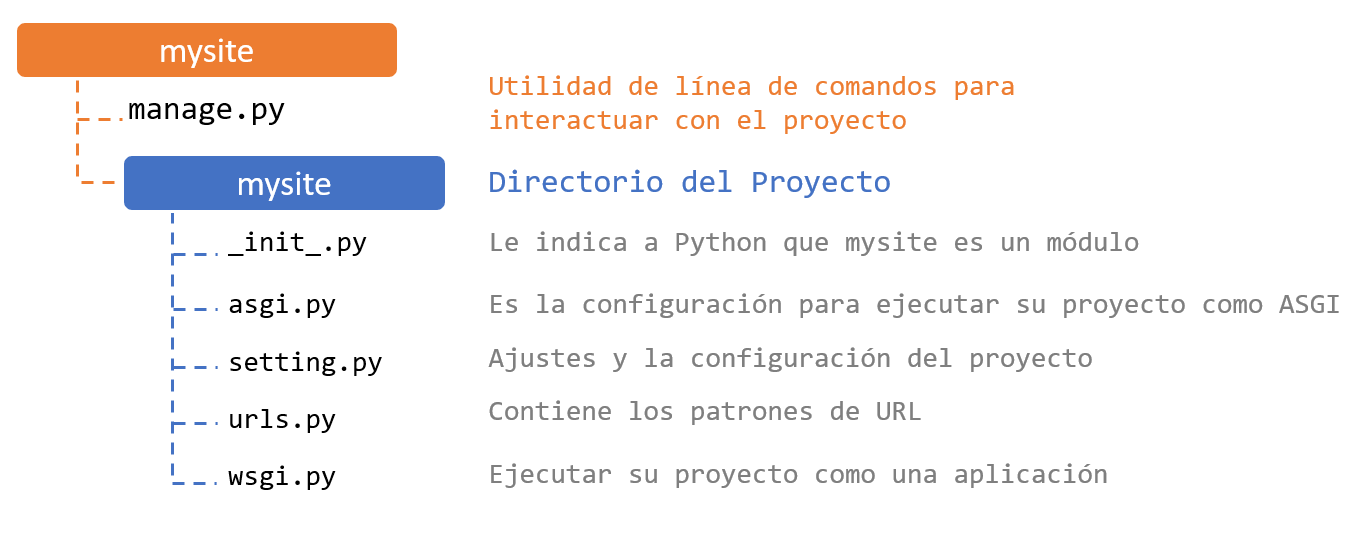
* 1. Salimos de Python con exit()

## Crear un Proyecto en Django

* 1. Nos ubicamos en el directorio de nuestro proyecto con cd C:\Users\user\Desktop\Proyectos\_Django
  2. Para crear el proyecto mysite, ejecutamos el comando: django-admin startproject mysite



* 1. Verifiquemos que la siguiente estructura del proyecto, ha sido creada:

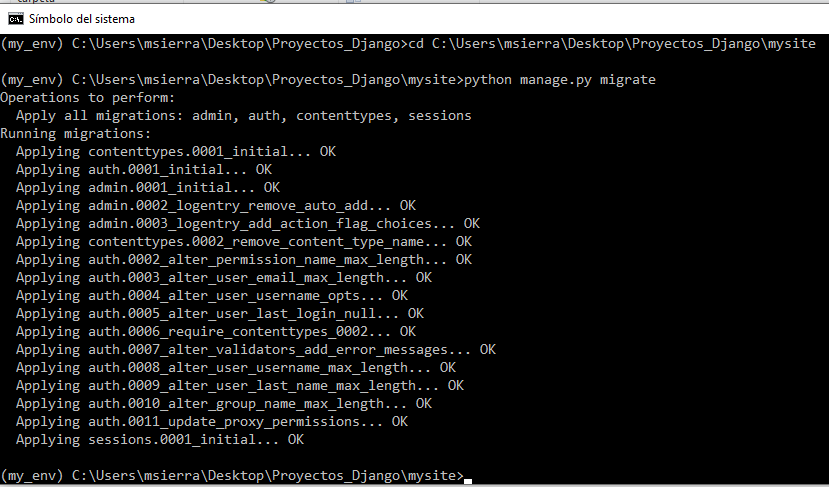


### Completar la configuración del Proyecto y crear las tablas asociadas a los modelos

1. Nos ubicamos en directorio principal de nuestro proyecto:

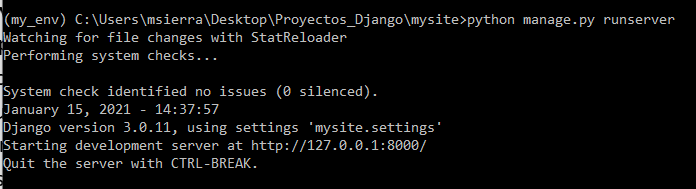
cd C:\Users\user\Desktop\Proyectos\_Django\mysite

1. Ejecutamos el comando: python manage.py migrate

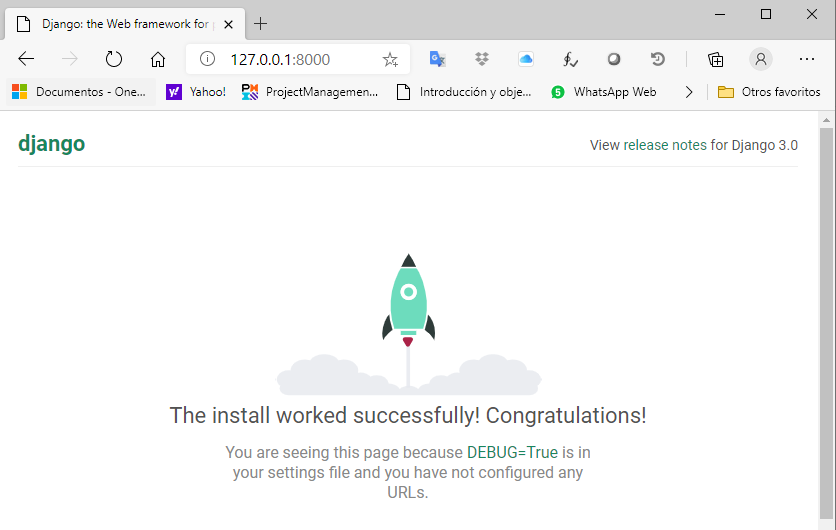


### Ejecutar el servidor de desarrollo

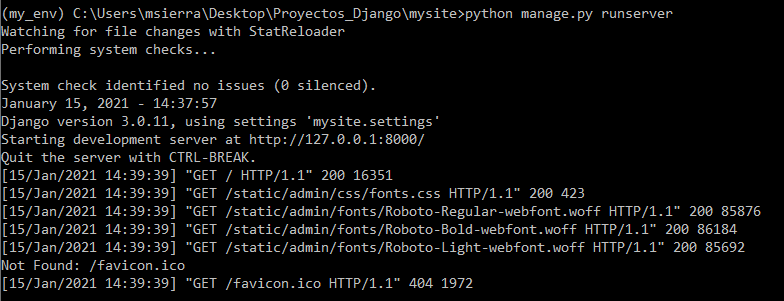
1. Iniciaremos el servidor de desarrollo escribiendo el siguiente comando desde la carpeta raíz de su proyecto: python manage.py runserver



1. Abrimos nuestro navegador y copiamos la URL <http://127.0.0.1:8000/>. La siguiente página nos indica que nuestro proyecto se está ejecutando correctamente.



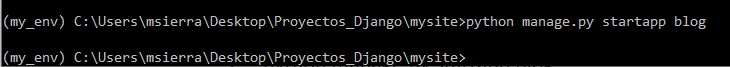
1. Si observamos la consola, vemos el requerimiento GET a la página



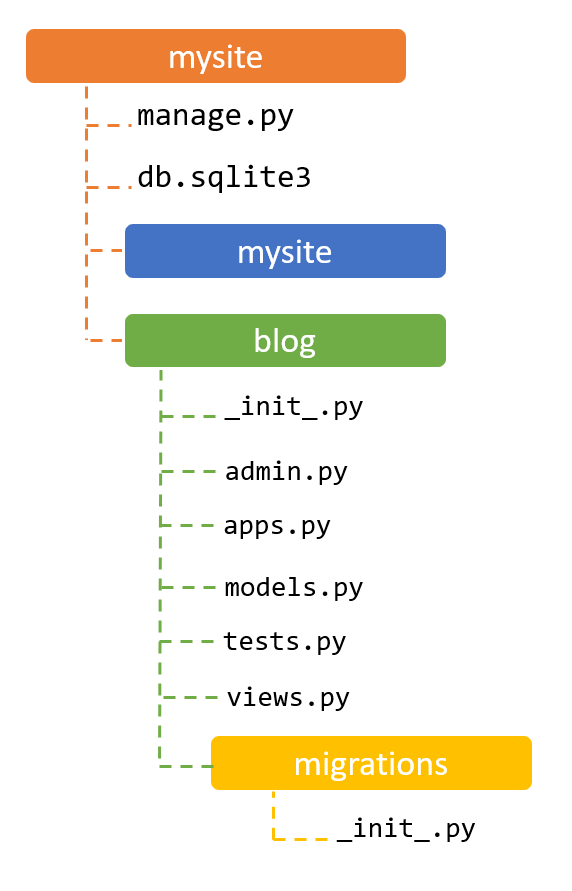
1. Utilicemos CTRL-C para salir del Servidor

## Crear una aplicación

* 1. Desde el directorio raíz del proyecto, ejecutamos el siguiente comando python manage.py startapp blog



* 1. Verificamos la estructura creada



### Diseñar el esquema de datos del blog

1. En el archivo models.py, definiremos el esquema de datos de nuestro blog.
2. Abrimos el archivo C:\Users\user\Desktop\Proyectos\_Django\mysite\blog\models.py con Pychar e insertamos el siguiente código:

from django.db import models

from django.utils import timezone

from django.contrib.auth.models import User

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,

on\_delete=models.CASCADE,

related\_name='blog\_posts')

body = models.TextField()

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

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

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

status = models.CharField(max\_length=10,

choices=STATUS\_CHOICES,

default='draft')

class Meta:

ordering = ('-publish',)

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

return self.title

1. Guardamos los cambios realizados

### Activando la aplicación

1. Editamos el archivo C:\Users\user\Desktop\Proyectos\_Django\mysite\mysite\settings.py e insertamos blog.apps.BlogConfig a la configuración INSTALLED\_APPS.

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

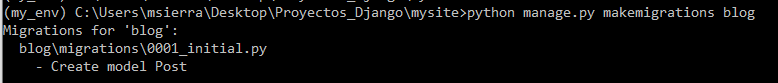
'django.contrib.messages',

'django.contrib.staticfiles',

'blog.apps.BlogConfig',]

### Crear y aplicar migraciones

1. El comando migrate aplica migraciones para todas las aplicaciones enumeradas en INSTALLED\_APPS; sincroniza la base de datos con los modelos actuales y las migraciones existentes. Desde el directorio raíz de nuestro proyecto, ejecutamos el siguiente comando: python manage.py makemigrations blog



1. Django acaba de crear el archivo 0001\_initial.py dentro del directorio migrations. Desde PyCharm podemos abrirlo y revisar el contenido creado.
2. Veamos el código SQL que Django ejecutará en la base de datos para crear la tabla nuestro modelo. Ejecutemos el siguiente comando para inspeccionar la salida SQL de su primera migración: python manage.py sqlmigrate blog 0001:

BEGIN;

--

-- Create model Post

--

CREATE TABLE "blog\_post" (

"id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,

"title" varchar(250) NOT NULL,

"slug" varchar(250) NOT NULL,

"body" text NOT NULL,

"publish" datetime NOT NULL,

"created" datetime NOT NULL,

"updated" datetime NOT NULL,

"status" varchar(10) NOT NULL,

"author\_id" integer NOT NULL REFERENCES

"auth\_user" ("id") DEFERRABLE INITIALLY DEFERRED);

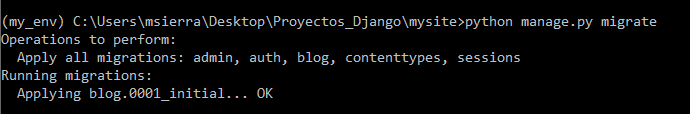
CREATE INDEX "blog\_post\_slug\_b95473f2" ON "blog\_post" ("slug");

CREATE INDEX "blog\_post\_author\_id\_dd7a8485"

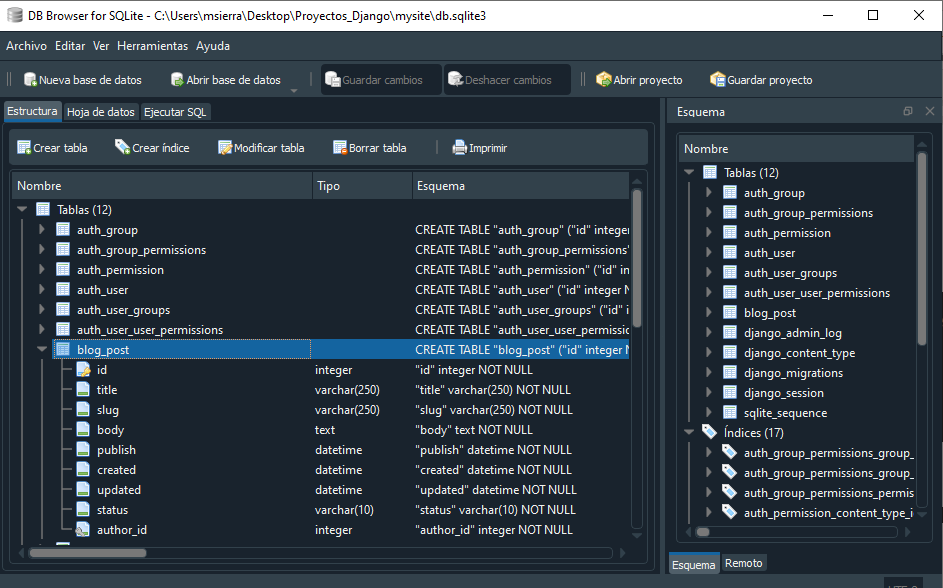
ON "blog\_post" ("author\_id");

COMMIT;

1. Sincronicemos nuestra base de datos con el nuevo modelo. Ejecutemos el siguiente comando para aplicar las migraciones existentes: python manage.py migrate.

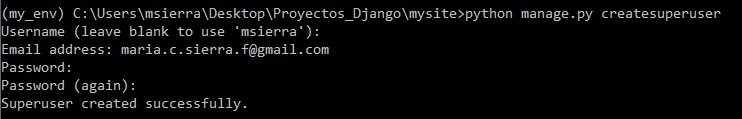


1. Utilizando el DB Browser for SQLite podemos ver la tabla creada para nuestra aplicación blog:

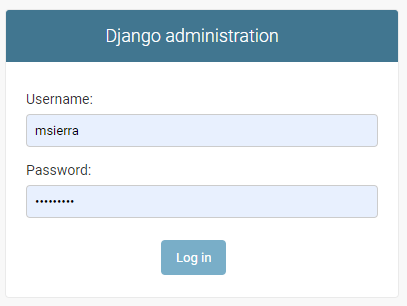


### Crear un usuario administrador

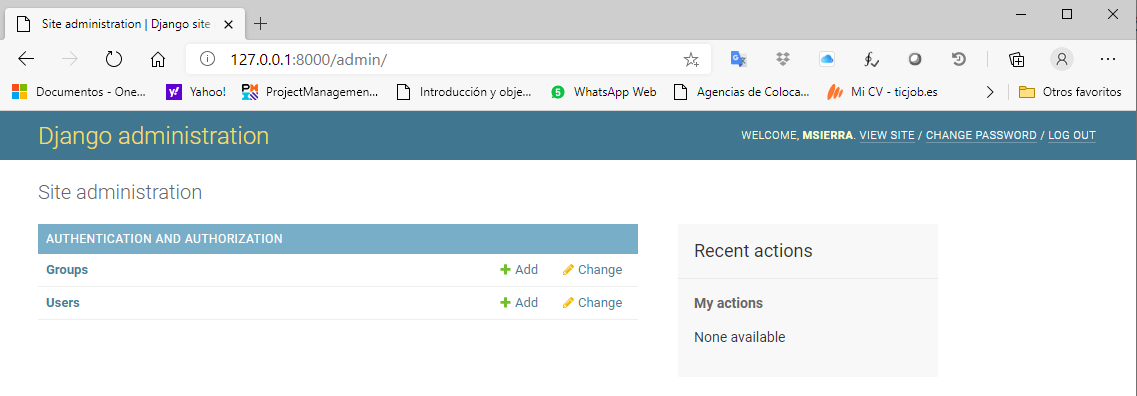
1. Para crear el usuario administrador, debemos ejecutar el siguiente comando: python manage.py createsuperuser.



1. Iniciamos el servidor de desarrollo con el comando python manage.py runserver
2. Abrimos la URL http://127.0.0.1:8000/admin/ en nuestro navegador



1. Introducimos las credenciales del usuario creado previamente. Veremos la página del sitio de administración:



### Agregar modelos al sitio de administración

1. Agreguemos nuestros modelos de blog al sitio de administración. Para ello, editamos el archivo C:\Users\user\Desktop\Proyectos\_Django\mysite\blog \admin.py de la aplicación de blog. Incluimos las siguiente líneas:

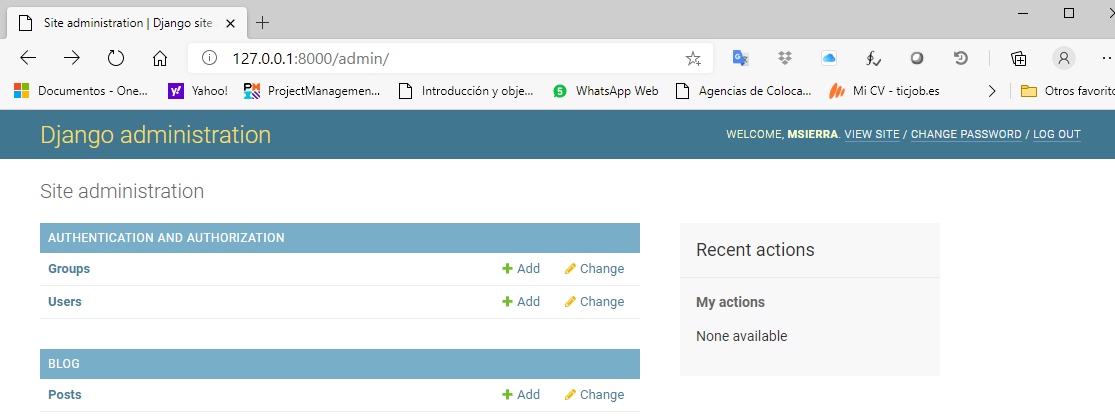
from django.contrib import admin

# Aplicación Blog

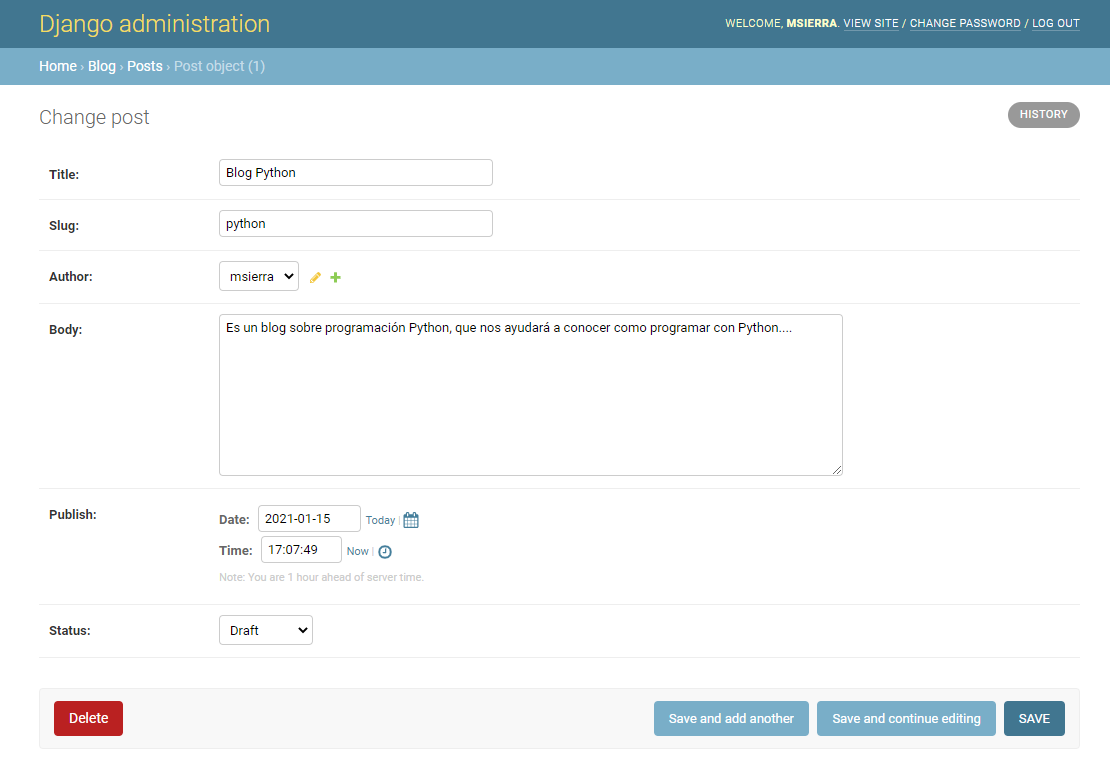
from .models import Post

admin.site.register(Post)

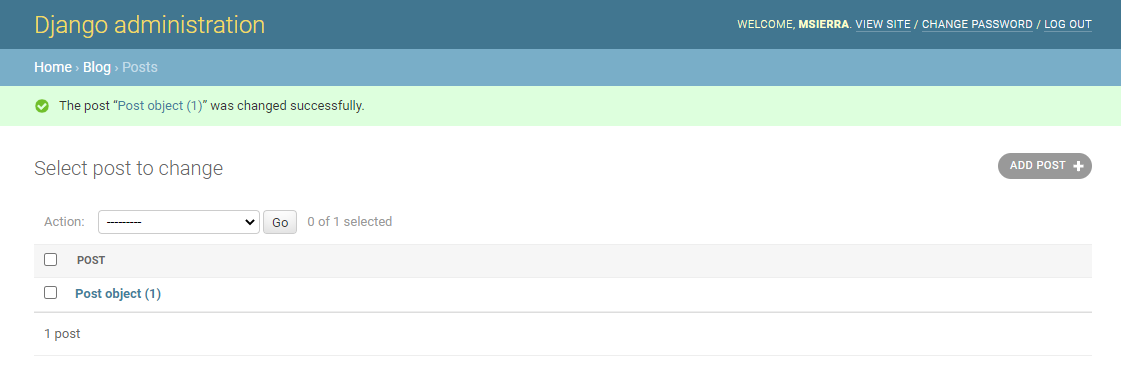
1. Volvemos a cargar el sitio de administración <http://127.0.0.1:8000/admin/>



1. Agregamos una nueva publicación, haciendo clic en el enlace +Add junto a Posts. Nos muestra el siguiente formulario:



1. Completamos el formulario y hacemos clic en el botón SAVE. Nos debe mostrar la siguiente pantalla:



### Modificar la página de administración de Posts

* 1. Editamos el archivo admin.py de nuestra aplicación de blog, insertando el siguiente código

from django.contrib import admin

from .models import Post

@admin.register(Post)

class PostAdmin(admin.ModelAdmin):

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

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

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

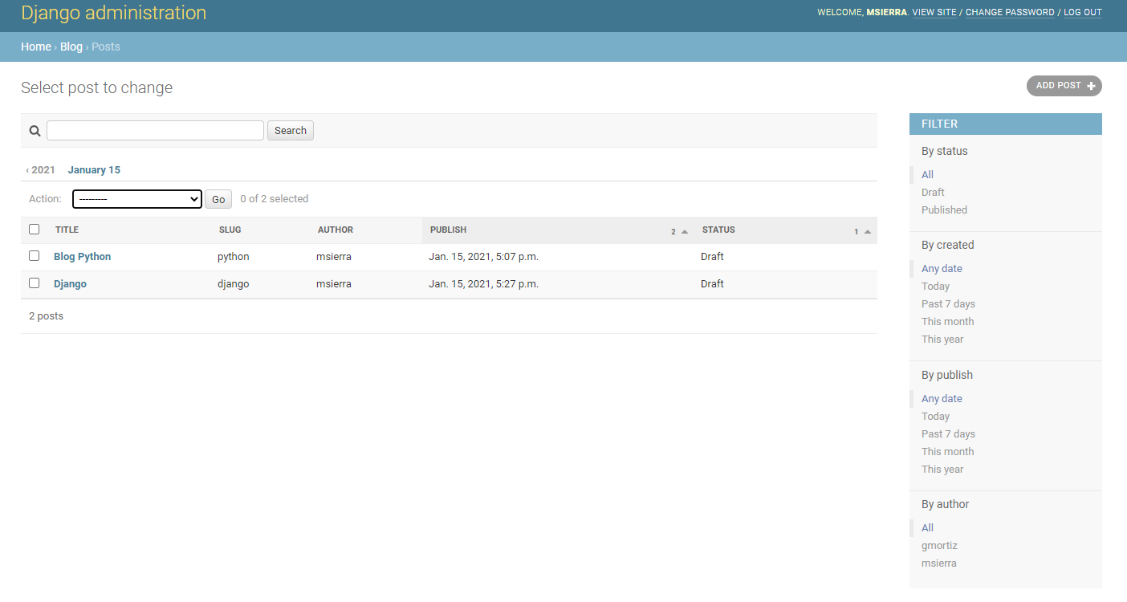
prepopulated\_fields = {'slug': ('title',)}

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

date\_hierarchy = 'publish'

ordering = ('status', 'publish')

* 1. Regresemos al navegador y cargamos nuevamente la página <http://127.0.0.1:8000/admin/> de la lista de publicaciones o Posts



### Creemos un Posts en memoria y luego en BBDD

1. Desde la terminal ejecutamos C:\Users\user\Desktop\Proyectos\_Django\mysite> python manage.py Shell.
2. Desde el prompt de Python escribimos:

>>> from django.contrib.auth.models import User

>>> from blog.models import Post

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

>>> post = Post(title='Base de Datos MongoDB',

... slug='base-datos-mongodb',

... body='La BBDD MongoDB es una base de datos documental.',

... author=user)

>>> post.save()

### Obtener todos los posts

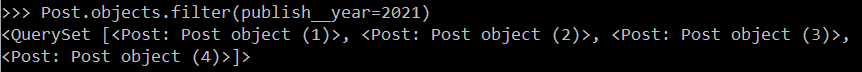
1. Desde el prompt de Python escribimos

>>> all\_posts = Post.objects.all()

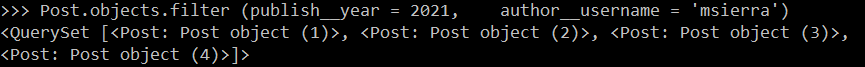
>>> all\_posts

1. Mostrar todos los posts del publicados en el 2021.

>>> Post.objects.filter (publish\_\_year = 2021)

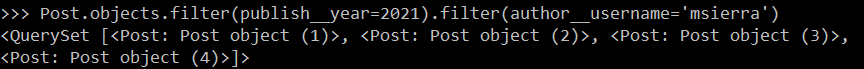


>>> Post.objects.filter (publish\_\_year = 2021, author\_\_username = 'user')



>>> Post.objects.filter(publish\_\_year=2021) \

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

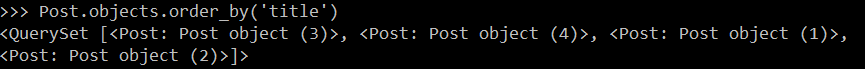


>>> Post.objects.filter(publish\_\_year=2021) \

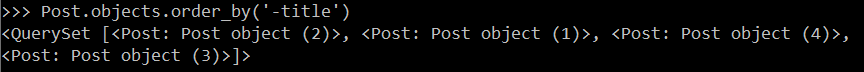
>>> .exclude(title\_\_startswith='Blog')



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



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



### Agregar un administrador personalizado

1. Editamos el archivo models.py, ubicado en C:\Users\user\Desktop\Proyectos\_Django\mysite\blog

from django.db import models

from django.utils import timezone

from django.contrib.auth.models import User

class PublishedManager(models.Manager):

def get\_queryset(self):

return super(PublishedManager,

self).get\_queryset().filter(status='published')

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,

on\_delete=models.CASCADE,

related\_name='blog\_posts')

body = models.TextField()

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

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

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

status = models.CharField(max\_length=10,

choices=STATUS\_CHOICES,

default='draft')

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

published = PublishedManager() # Our custom manager.

class Meta:

ordering = ('-publish',)

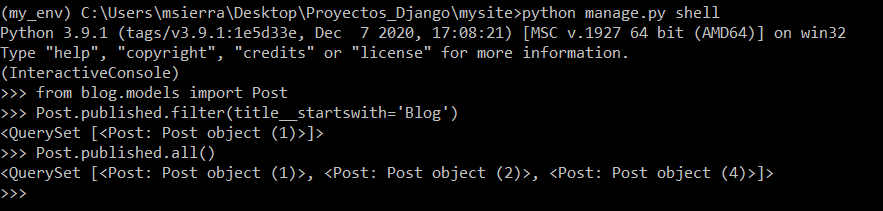
def \_\_str\_\_(self):

return self.title

1. Para obtener resultados para este QuerySet, debemos cambiar el campo status a Published en el objeto Post cuyo título comienza con Blog.
2. Iniciamos el servidor de desarrollo nuevamente con el siguiente comando: C:\Users\user\Desktop\Proyectos\_Django\mysite>python manage.py Shell.
3. Importamos el modelo Post y recuperamos todas las publicaciones cuyo título comience con Blog, ejecutando el siguiente QuerySet:

>>> from blog.models import Post

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



### Crear vistas de lista y detalle

1. Comencemos creando una vista para mostrar la lista de publicaciones. Editemos el archivo views.py de nuestra aplicación de blog:

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

from .models import Post

def post\_list(request):

posts = Post.published.all()

return render(request,

'blog/post/list.html',

{'posts': posts})

1. Creamos una segunda vista que muestra una sola publicación

def post\_detail(request, year, month, day, post):

post = get\_object\_or\_404(Post, slug=post,

status='published',

publish\_\_year=year,

publish\_\_month=month,

publish\_\_day=day)

return render(request,

'blog/post/detail.html',

{'post': post})

1. Creamos un archivo urls.py en la carpeta blog

from django.urls import path

from . import views

app\_name = 'blog'

urlpatterns = [

# post views

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

path('<int:year>/<int:month>/<int:day>/<slug:post>/',

views.post\_detail,

name='post\_detail'),

]

1. Editamos el archivo urls.py en la carpeta mysite

from django.urls import path, include

from django.contrib import admin

urlpatterns = [

path('admin/', admin.site.urls),

path('blog/', include('blog.urls', namespace='blog')),

]

1. Utilizaremos la URL post\_detail para crear la URL canónica para los objetos de publicación. Editemos el archivo models.py de la aplicación de blog y agreguemos el siguiente código:

from django.urls import reverse

class Post(models.Model):

# ...

def get\_absolute\_url(self):

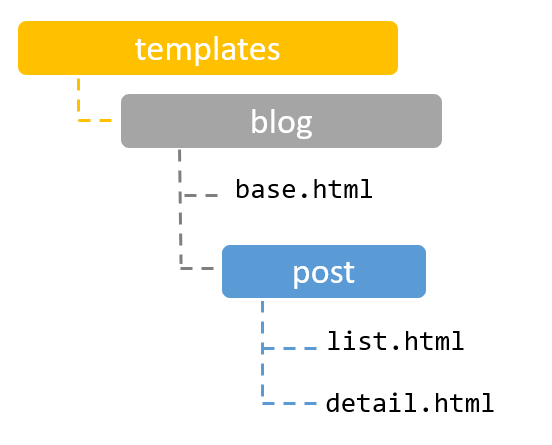
return reverse('blog:post\_detail',

args=[self.publish.year,

self.publish.month,

self.publish.day, self.slug])

1. Agreguemos plantillas a la aplicación para mostrar publicaciones de una manera fácil de usar. Creemos los siguientes directorios y archivos dentro del directorio de su aplicación de blog:



1. Editemos el archivo base.html:

{% load static %}

<!DOCTYPE html>

<html>

<head>

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

<link rel="stylesheet" type="text/css"

href="/static/admin/css/base.css">

</head>

<body>

<div id="content">

{% block content %}

{% endblock %}

</div>

<div id="sidebar">

<h2>Blog de publicaciones en Django</h2>

<p>Curso de Python.</p>

</div>

</body>

</html>

1. Editemos el archivo post/list.html:

{% extends "blog/base.html" %}

{% block title %}My Blog{% endblock %}

{% block content %}

<h1>Blog de Publicaciones en Django</h1>

{% for post in posts %}

<h2>

<a href="{{ post.get\_absolute\_url }}">

{{ post.title }}

</a>

</h2>

<p class="date">

Published {{ post.publish }} by {{ post.author }}

</p>

{{ post.body|truncatewords:30|linebreaks }}

{% endfor %}

{% endblock %}

1. Abrimos el Shell desde C:\Users\user\Desktop\Proyectos\_Django\mysite y ejecute el comando: python manage.py runserver.
2. Abrimos la URL del nuestro sitio http://127.0.0.1:8000/blog/.



1. Editamos el archivo post/detail.html, para visualizar el detalle de las publicaciones:

{% extends "blog/base.html" %}

{% block title %}{{ post.title }}{% endblock %}

{% block content %}

<h1>{{ post.title }}</h1>

<p class="date">

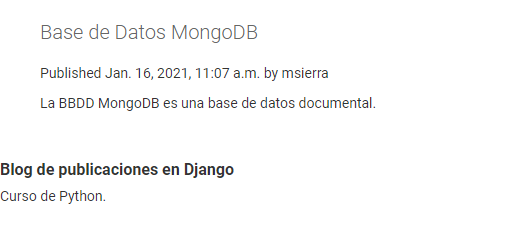
Published {{ post.publish }} by {{ post.author }}

</p>

{{ post.body|linebreaks }}

{% endblock %}

1. Abrimos la URL del nuestro sitio <http://127.0.0.1:8000/blog/> y seleccionemos una publicación.



1. Agregamos páginación: Editemos el archivo views.py de la aplicación de blog para importar las clases del “paginador” de Django y modificamos la vista post\_list, de la siguiente manera:

from django.core.paginator import Paginator, EmptyPage,\

PageNotAnInteger

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})

1. Crear una plantilla para mostrar el “paginador” de modo que pueda incluirse en cualquier plantilla que utilice la paginación. En la carpeta templates/ de la aplicación de blog, creamos un nuevo archivo pagination.html. Agregue el siguiente código HTML al archivo:

<div class="pagination">

<span class="step-links">

{% if page.has\_previous %}

<a href="?page={{ page.previous\_page\_number }}">Previous</a>

{% endif %}

<span class="current">

Page {{ page.number }} of {{ page.paginator.num\_pages }}.

</span>

{% if page.has\_next %}

<a href="?page={{ page.next\_page\_number }}">Next</a>

{% endif %}

</span>

</div>

1. Regresemos a la plantilla blog/post/list.html e incluyamos la plantilla pagination.html en la parte inferior del bloque {% content%}, de la siguiente manera:

{% block content %}

...

{% include "pagination.html" with page=posts %}

{% endblock %}

1. Abrimos nuestra página http://127.0.0.1:8000/blog/ en el navegador. Deberíamos ver la paginación en la parte inferior de la lista de publicaciones y debería poder navegar por las páginas:



## Funciones Avanzadas

### Compartir publicaciones por Correo Electrónico

* 1. Creamos un archivo forms.py dentro del directorio de la aplicación de blog e insertemos el siguiente código:

from django import forms

class EmailPostForm(forms.Form):

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

email = forms.EmailField()

to = forms.EmailField()

comments = forms.CharField(required=False,

widget=forms.Textarea)

* 1. Debemos crear una nueva vista que maneje el formulario y envíe un correo electrónico correctamente. Para ello, debemos editar el archivo views.py de la aplicación de blog insertamos el siguiente código:

from .forms import EmailPostForm

def post\_share(request, post\_id):

# Retrieve post by id

post = get\_object\_or\_404(Post, id=post\_id, status='published')

if request.method == 'POST':

# Form was submitted

form = EmailPostForm(request.POST)

if form.is\_valid():

# Form fields passed validation

cd = form.cleaned\_data

# ... send email

else:

form = EmailPostForm()

return render(request, 'blog/post/share.html',

{'post': post,'form': form})

* 1. Vamos a configurar un correo gmail, para enviar correos agregando la siguiente configuración al archivo settings.py

# Este código es para utiliza la consola

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

EMAIL\_HOST = 'smtp.gmail.com'  
EMAIL\_HOST\_USER ='curso.python.cloud@gmail.com'  
EMAIL\_HOST\_PASSWORD ='cursopython'  
EMAIL\_PORT = 587  
EMAIL\_USE\_TLS = True

* 1. Realicemos pruebas, para validar que funciona el correo. Ejecutamos python manage.py shell para abrir el shell de Python y enviar un correo electrónico, de la siguiente manera:

from django.core.mail import send\_mail

send\_mail('Django mail', 'Este email ha sido enviado desde Django.', 'curso.python.cloud@gmail.com', ['curso.python.cloud@gmail.com'], fail\_silently=False)

* 1. Editamos la vista post\_share en el archivo views.py de la aplicación de blog, de la siguiente manera:

def post\_share(request, post\_id):  
 # Retrieve post by id  
 post = get\_object\_or\_404(Post, id=post\_id, status='published')  
 sent = False  
 if request.method == 'POST':  
 # Form was submitted  
 form = EmailPostForm(request.POST)  
 if form.is\_valid():  
 # Form fields passed validation  
 cd = form.cleaned\_data  
 post\_url = request.build\_absolute\_uri(  
 post.get\_absolute\_url())  
 subject = f"{cd['name']} recommends you read " \  
 f"{post.title}"  
 message = f"Read {post.title} at {post\_url}\n\n" \  
 f"{cd['name']}\'s comments: {cd['comments']}"  
 send\_mail(subject, message, 'curso.python.cloud@gmail.com',  
 [cd['to']])  
 sent = True  
 else:  
 form = EmailPostForm()  
 return render(request, 'blog/post/share.html', {'post': post,  
 'form': form,  
 'sent': sent})

* 1. Ahora que la vista está completa, debemos agregar un nuevo patrón de URL. Para ello, abrimos el archivo urls.py de la aplicación de blog (C:\Users\user\Desktop\Proyectos\_Django\mysite\blog) y agregamos el patrón de URL post\_share, de la siguiente manera:

urlpatterns = [

# ...

path('<int:post\_id>/share/',

views.post\_share, name='post\_share'),

]

* 1. Después de crear el formulario, la vista y agregar el patrón de URL, solo le falta la plantilla para esta vista. Creemos un nuevo archivo en el directorio blog/templates/blog/post/ con el nombre share.html. Agregue el siguiente código:

{% extends "blog/base.html" %}

{% block title %}Share a post{% endblock %}

{% block content %}

{% if sent %}

<h1>E-mail successfully sent</h1>

<p>

"{{ post.title }}" was successfully sent to {{ form.cleaned\_data.to }}.

</p>

{% else %}

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

<form method="post">

{{ form.as\_p }}

{% csrf\_token %}

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

</form>

{% endif %}

{% endblock %}

* 1. Editamos la plantilla blog/post/detail.html e insertamos el siguiente enlace a la URL de publicación compartida después de la variable {{post.body | linebreaks}}:

<p>

<a href="{% url "blog:post\_share" post.id %}">

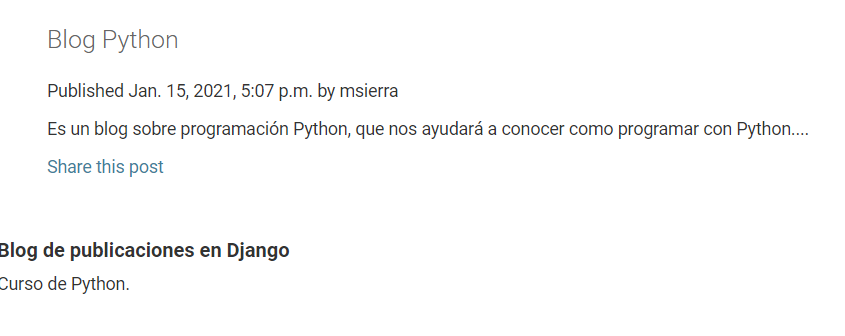
Share this post

</a>

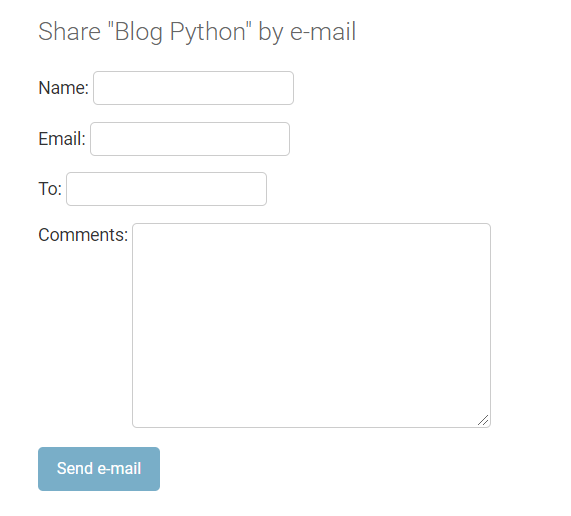
</p>

* 1. Ahora podemos ver el resultado final:

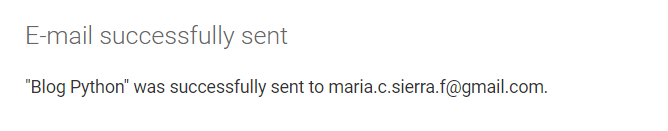
1. Iniciamos el servidor de desarrollo con el comando python manage.py runserver.
2. Abrimos nuestra aplicación http://127.0.0.1:8000/blog/ en el navegador.
3. Seleccionamos el título de cualquier publicación para ver su página de detalles. Observamos, debajo del cuerpo de la publicación, el enlace “[Share this post](http://127.0.0.1:8000/blog/1/share/)”



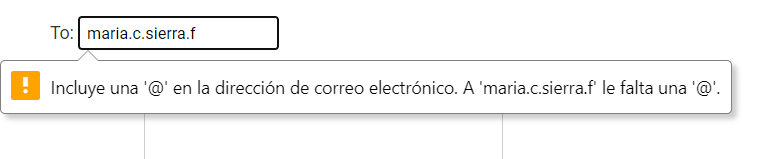
1. Al seleccionar el enlace “[Share this post](http://127.0.0.1:8000/blog/1/share/)”, se abre una nueva página con el formulario para compartir esta publicación por correo electrónico:



1. Al seleccionar el botón “Send e-mail”, el formulario se envía y se valida. Si todos los campos contienen datos válidos, recibiremos un mensaje de éxito, como se indica a continuación:



1. Si ingresamos datos no válidos, el formulario se procesa nuevamente, incluidos todos los errores de validación:



### Crear el modelo de comentarios

* 1. Abrimos el archivo models.py de la aplicación de blog e insertamos el siguiente código:

class Comment(models.Model):

post = models.ForeignKey(Post,

on\_delete=models.CASCADE,

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)

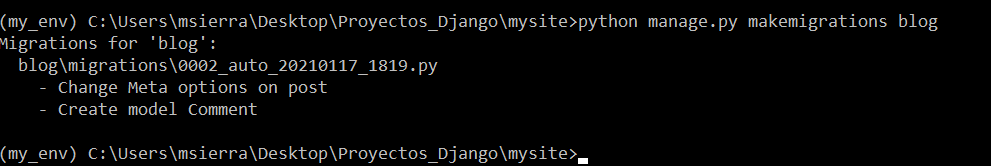
class Meta:

ordering = ('created',)

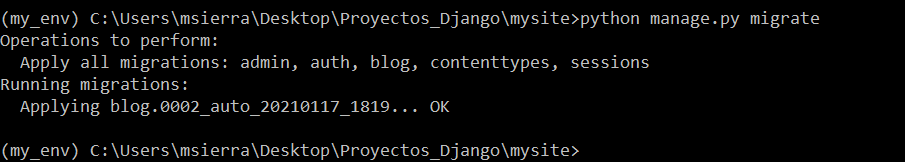
def \_\_str\_\_(self):

return f'Comment by {self.name} on {self.post}'

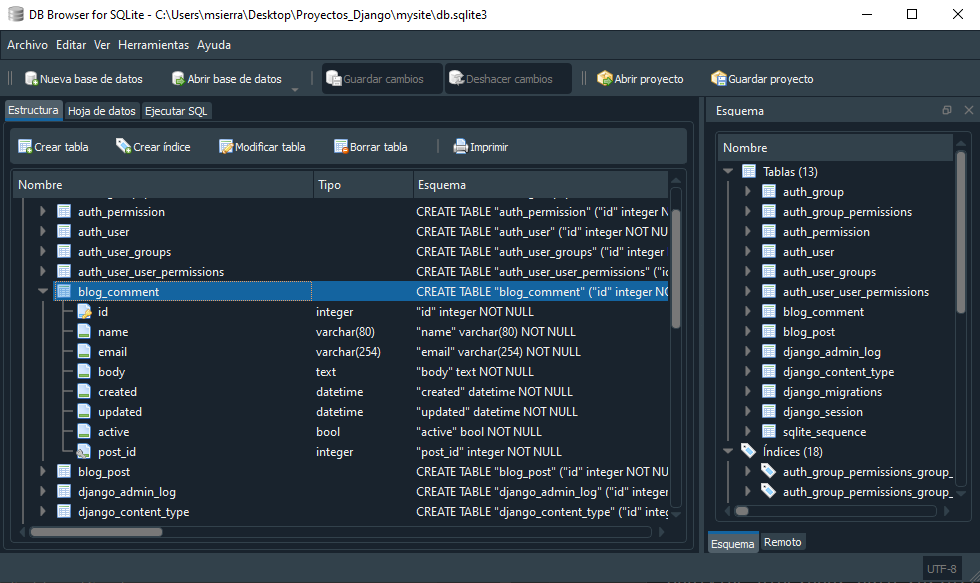
* 1. El nuevo modelo Comment creado aún no está sincronizado con la base de datos. Debemos ejecutar el siguiente comando para generar una nueva migración que refleje la creación del nuevo modelo: (my\_env) C:\Users\user\Desktop\Proyectos\_Django\mysite> python manage.py makemigrations blog.



* 1. Django ha generado un archivo 0002\_comment.py dentro del directorio migrations/ de la aplicación del blog.
  2. Ahora, debemos crear el esquema de base de datos relacionado y aplicar los cambios a la base de datos. Debemos ejecutar el siguiente comando para aplicar las migraciones existentes: python manage.py migrate.



* 1. Ahora existe una tabla blog\_comment en la base de datos.



* 1. A continuación, podemos agregar el nuevo modelo al sitio de administración para gestionar los comentarios a través de una interfaz simple. Abrimos el archivo admin.py e insertamos la siguiente clase ModelAdmin:

from .models import Post, Comment

**@**admin**.register(**Comment**)**

class **CommentAdmin(**admin**.ModelAdmin):**

**list\_display = (**'name'**,** 'email'**,** 'post'**,** 'created'**,** 'active'**)**

**list\_filter = (**'active'**,** 'created'**,** 'updated'**)**

**search\_fields = (**'name'**,** 'email'**,** 'body'**)**

* 1. Crear un formulario para enviar comentarios dinámicamente desde el modelo, editamos el archivo forms.py de la aplicación de blog e insertamos las siguientes líneas:

from .models import Comment

class CommentForm(forms.ModelForm):

class Meta:

model = Comment

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

* 1. Utilizaremos la vista de detalles de la publicación para crear una instancia del formulario y procesarlo. Editamos el archivo views.py:

from .models import Post, Comment

from .forms import EmailPostForm, CommentForm

def post\_detail(request, year, month, day, post):

post = get\_object\_or\_404(Post, slug=post,

status='published',

publish\_\_year=year,

publish\_\_month=month,

publish\_\_day=day)

# List of active comments for this post

comments = post.comments.filter(active=True)

new\_comment = None

if request.method == 'POST':

# A comment was posted

comment\_form = CommentForm(data=request.POST)

if comment\_form.is\_valid():

# Create Comment object but don't save to database yet

new\_comment = comment\_form.save(commit=False)

# Assign the current post to the comment

new\_comment.post = post

# Save the comment to the database

new\_comment.save()

else:

comment\_form = CommentForm()

return render(request,

'blog/post/detail.html',

{'post': post,

'comments': comments,

'new\_comment': new\_comment,

'comment\_form': comment\_form})

* 1. Para mostrar el número total de comentarios de una publicación. Abrimos la plantilla post/detail.html e insertamos el siguiente código al bloque de contenido:

{% with comments.count as total\_comments %}

<h2>

{{ total\_comments }} comment{{ total\_comments|pluralize }}

</h2>

{% endwith %}

{% for comment in comments %}

<div class="comment">

<p class="info">

Comment {{ forloop.counter }} by {{ comment.name }}

{{ comment.created }}

</p>

{{ comment.body|linebreaks }}

</div>

{% empty %}

<p>There are no comments yet.</p>

{% endfor %}

{% if new\_comment %}

<h2>Your comment has been added.</h2>

{% else %}

<h2>Add a new comment</h2>

<form method="post">

{{ comment\_form.as\_p }}

{% csrf\_token %}

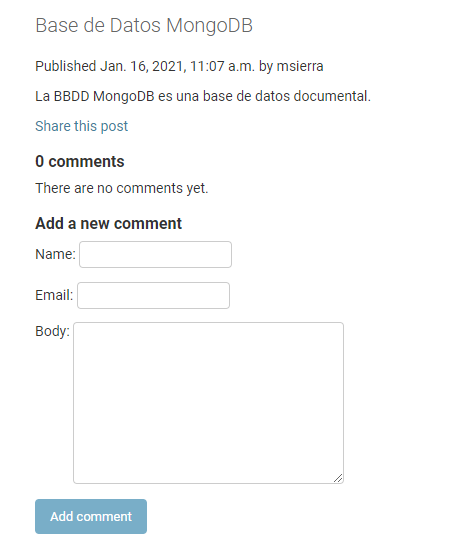
<p><input type="submit" value="Add comment"></p>

</form>

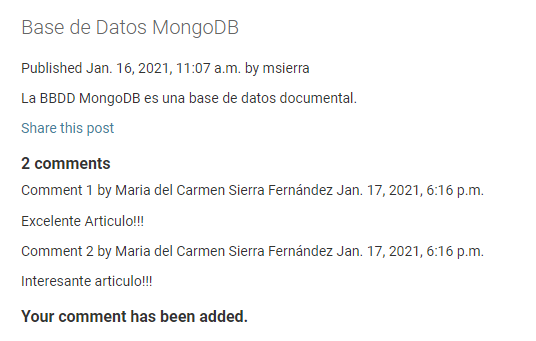
{% endif %}

* 1. Veamos los resultados:

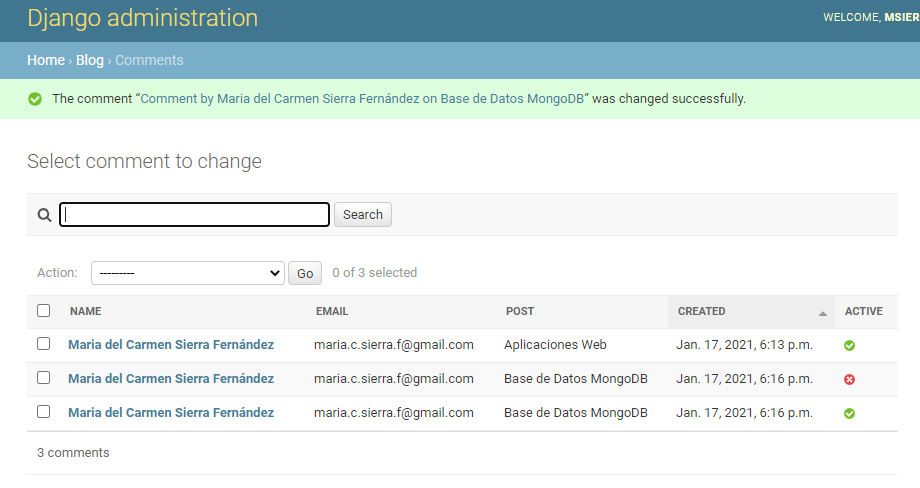
1. Iniciamos el servidor de desarrollo con el comando python manage.py runserver.
2. Abrimos nuestra aplicación http://127.0.0.1:8000/blog/ en el navegador y seleccionamos el título de una publicación para ver su página de detalles.



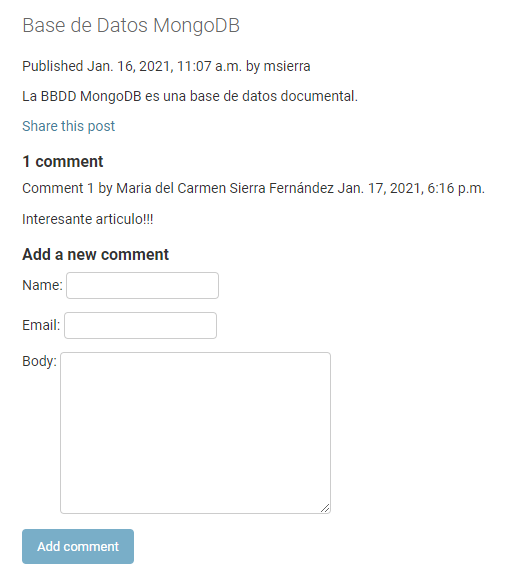
1. Agreguemos un par de comentarios usando el formulario. Deben aparecer debajo de su publicación en orden cronológico, de la siguiente manera:



1. Si abrimos la página http://127.0.0.1:8000/admin/blog/comment/ en el navegador. Veremos la página de administración con la lista de comentarios que acabamos de crear.
2. Seleccionemos el nombre de uno de los comentarios para editarlo, desmarquemos la casilla de verificación Active y seleccionemos el botón SAVE. Seremos redirigido a la lista de comentarios nuevamente, y la columna ACTIVE mostrará un ícono inactivo para el comentario.



1. Si regresamos a la vista de detalles de la publicación, notaremos que el comentario inactivo ya no se muestra; tampoco lo considera para el número total de comentarios. El campo active, nos permite desactivar los comentarios inapropiados y evitar mostrarlos en las publicaciones.



## Como levantar la Página Web en desarrollo:

1. Ubicarse en el directorio cd C:\Users\msierra\Desktop\Proyectos\_Django\my\_env\Scripts
2. Ejecutar el fichero activate.bat, para activar el entorno
3. Ubicarse en el directorio: cd C:\Users\msierra \Desktop\Proyectos\_Django\mysite
4. Levantar el servidor: python manage.py runserver
5. Abrir la página http://127.0.0.1:8000/