PROYECTO FINAL - INTEGRANDO TODO LO DADO

Python Django & MySql

Damos inicio al final

En este proyecto final vamos a integrar todos los conocimientos adquiridos a lo largo del curso para confeccionar una web final usando DJANGO como framework

POLO IT /CHACO
UNCAUS Universidad Nacional del Chaco Austral
Globant
Universidad Tecnológica Nacional Facultad Regional Resistencia
INFORMATORIO

Subsecretaría de Empleo
Ministerio de Producción, Industria y Empleo
CHACO Gobierno de todos

CREANDO EL ENTORNO VIRTUAL

¿QUÉ ES EL ENTORNO VIRTUAL?

Imagina que estás trabajando en tres proyectos diferentes en un taller. En el taller, tienes 50 herramientas diferentes que podrías necesitar para los proyectos, como destornilladores, martillos, llaves inglesas, etc.

Si el taller está desordenado y todas las herramientas están mezcladas, sería difícil encontrar las herramientas específicas que necesitas para cada proyecto. Tendrías que buscar en toda la sala y perderías tiempo valioso tratando de ubicar las herramientas correctas.

Pero, si el taller está organizado en sectores, donde cada sector corresponde a un proyecto en particular, y las herramientas necesarias para cada proyecto se colocan en su respectivo sector, la situación mejora. Cada sector del taller tendría las herramientas específicas que necesitas para ese proyecto en particular.

Además, hay un sector neutro donde se colocan las herramientas que se comparten entre los proyectos. Estas herramientas son utilizadas por varios proyectos y se mantienen en un lugar central para que todos los proyectos puedan acceder fácilmente a ellas.

Con este enfoque, puedes ir al sector correspondiente al proyecto en el que estás trabajando y encontrar todas las herramientas necesarias allí mismo, sin tener que buscar en otros lugares. También puedes consultar el sector neutro cuando necesites herramientas compartidas.

En el desarrollo de software, los proyectos serían tus diferentes aplicaciones o sistemas, y las herramientas serían las bibliotecas, paquetes y configuraciones específicas para cada proyecto. Un entorno virtual sería como tener sectores ordenados en el taller, donde cada sector representa un proyecto y contienelas herramientas necesarias para ese proyecto. El sector neutro sería utilizado para herramientas compartidas que son utilizadas por varios proyectos.

De esta manera, los entornos virtuales te permiten trabajar en varios proyectos de manera organizada y eficiente, asegurándote de tener todas las herramientas necesarias disponibles sin confusiones ni conflictos entre los proyectos.

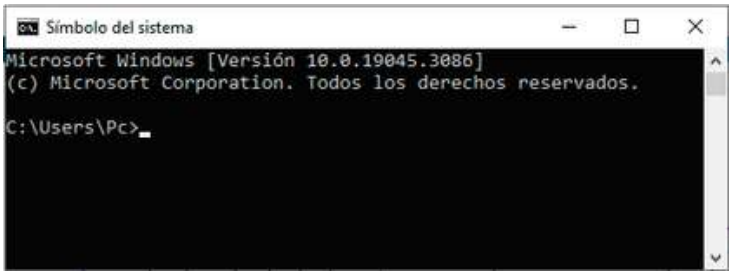
Entonces podemos decir que, un entorno virtual es como tener un taller organizado en sectores, donde cada sector representa un proyecto y contiene las herramientas específicas para ese proyecto. Esto facilita el acceso a las herramientas correctas sin tener que buscar en todas partes, y también proporciona un espacio central para las herramientas compartidas.

PASOS PARA LA CREACIÓN DEL ENTORNO

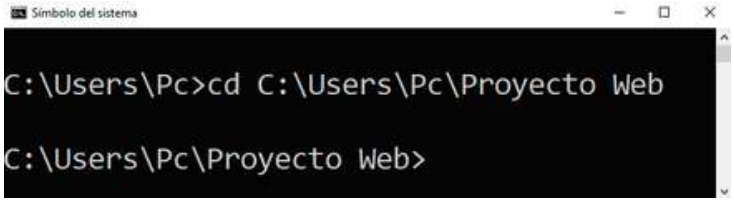
PASOS

Ahora que ya sabes lo que es un entorno virtual, vamos a crearlo (ya lo habrás visto en clase, pero te lo dejamos plasmado desde aquí):

1 - Trabajando desde Windows, vamos a abrir una consola (CMD).



Lo que debes hacer ahora es ubicarte en la carpeta donde vas a crear este proyecto final. Esto lo puedes hacer usando el comando cd, seguido de la ubicación donde estará tu proyecto final, o en este caso, donde vas a crear tu entorno virtual.



Utiliza el siguiente comando para instalar la paquetería necesaria para crear el entorno:

- **pip install virtualenv** (este comando te servirá para instalar la paquetería necesaria para crear tus entornos virtuales).

Procedemos a la creación del entorno:

- **python -m venv entorno** (este comando te servirá para crear tu entorno virutal, donde "entorno" es el nombre que nosotros escogimos para el entorno, puedes escoger cualquier nombre).

Ya tenemos nuestro entorno creado.
Siguiendo en la misma ubicación, vamos a movernos a la carpeta que se creó que es la de nuestro entorno, utilizando el comando cd seguido del nombre del entorno creado:

PASOS PARA INSTALAR DJANGO

PASOS

Si lo miras a través del explorador de Windows, podrás ver que esta carpeta se creó a partir del comando que utilizamos para crear el entorno. Dentro de esta carpeta se encuentran los archivos necesarios para nuestro entorno, sin embargo, para instalar Django (y el resto de paquetes que hagan falta), debemos activar nuestro entorno. Para ello usamos el comando cd nuevamente y nos dirigimos a la carpeta **Scripts:**

Dentro de esta carpeta procedemos a activar nuestro entorno para poder comenzar con la instalación de nuestros paquetes y comenzar a trabajar en el proyecto final. Usamos el comando **activate:**

Notarás como al principio de la línea de comando ahora se encuentra entre paréntesis, el nombre de nuestro entorno. Esto quiere decir que nuestro entorno está activado y listo para comenzar a trabajar en él. A modo de ordenar mejor todo, vamos a movernos a la carpeta raíz utilizando el comando cd.. (palabra cd y dos puntos). Lo haremos 2 veces para llegar a la carpeta principal:

Llega el momento de instalar Django por lo que usamos el comando **pip install django:**

COMENZAMOS NUESTRO PROYECTO

>>> PASOS

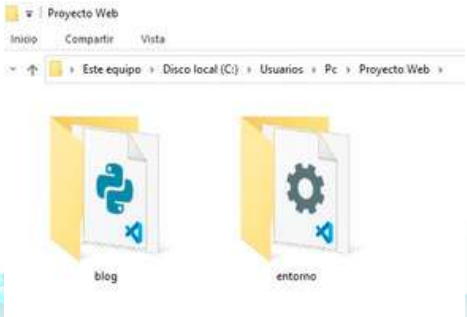
Django ya se encuentra instalado en nuestro entorno y podremos comenzar a trabajar con él.

*** si te aparecen esos mensajes de "notice" al final de la instalación de Django, no te preocupes, solo se está avisando que la versión de **pip** que estamos utilizando es una versión anterior a la actual, y nos dice que si queremos actualizar a la nueva versión debemos usar los comandos escritos en verde.
Lo puedes actualizar o seguirlo usando como lo vienes haciendo hasta ahora.***

Ya estamos listos para comenzar nuestro proyecto final y para eso vamos a utilizar el comando:

django-admin startproject blog (en este caso usamos el nombre "blog" para nuestro proyecto, pero puedes usar el nombre que quieras):

Hecho esto, ya está creado nuestro primer proyecto y ahora si estamos listos para comenzar a trabajar. Si abres un explorador de Windows podrás visualizar mejor la estructura que tenemos, en la cual tenemos la carpeta "entorno" que creamos a partir del comando **pip -m venv entorno** y ahora la carpeta "blog" a partir del comando para iniciar nuestro proyecto **django-admin startproject blog**



ESTRUCTURANDO NUESTRO PROYECTO

PASOS

Si accedemos a la carpta blog que se creó para nuestro proyecto, vamos a ver tenemos un archivo **manage.py** y otra carpeta llamada **blog**.

El archivo manage.py va a ser uno de los archivos funamentales que vamos a usar ya que con el podremos correr nuestro servidor, entre otras muchas funciones que nos brindará.

correr el servidor: nos referimos a que se generará un servidor local web con el cual vamos a poder ver el progreso de nuestra página a medida que la vamos modificando, como si estuvieramos navegando por internet, pero en realidad solo vamos a estar trabajando de manera local en nuestra pc, es decir, si durante este tiempo (hasta la instalación de otros paquetes) no tienes conexión a internet, de todas formas podrás ver tu página web y seguir trabajando en ella.

Si accedemos a la carpeta "blog" veremos que se encuentran otros archivos dentro de ella.

- El archivo **__init__.py** es un archivo especial en Python que se utiliza para indicar que un directorio es un paquete de Python. Aunque el archivo puede estar vacío, su presencia es esencial para que Python reconozca el directorio como un paquete válido.

- El archivo **asgi.py** es un archivo que se encuentra en un proyecto Django y se utiliza para configurar y ejecutar el servidor ASGI (Asynchronous Server Gateway Interface). ASGI es una especificación que permite que las aplicaciones web en Python se comuniquen con servidores web de manera asincrónica, lo que las hace adecuadas para manejar solicitudes concurrentes en tiempo real.

- El archivo **settings.py** es un archivo importante en un proyecto Django. Contiene la configuración principal de tu aplicación, como la base de datos, la configuración de aplicaciones, las claves secretas, las rutas de archivos estáticos y más. Esencialmente, el archivo settings.py controla el comportamiento y la apariencia de tu proyecto Django.

ESTRUCTURANDO NUESTRO PROYECTO

PASOS

- El archivo **urls.py** es un archivo de configuración clave en un proyecto Django. Define las rutas de URL de tu aplicación y cómo se relacionan con las vistas correspondientes. En otras palabras, el archivo urls.py mapea las URL que los usuarios ingresan en su navegador a las funciones o clases de vistas que deben manejar esas solicitudes.
- El archivo **wsgi.py** es un archivo de configuración utilizado en proyectos Django que implementan el estándar WSGI (Web Server Gateway Interface). WSGI es una especificación que define cómo las aplicaciones web en Python se comunican con los servidores web.

Vamos comenzar nuestro proyecto para trabajar de forma más ordenada, sin embargo, debemos aclarar que esto es solo una manera de hacerlo y lo hacemos por convención; si quieres realizarlo de otra manera, puedes hacerlo, pero, a fines prácticos para este proyecto, te recomendamos seguir la estructura que usaremos para que veas el paso a paso que iremos haciendo y puedas seguirnos de la misma forma.

Vamos a posicionarnos en la carpeta "blog" donde encontramos el archivo manage.py y la subcarpeta "blog".

Una vez posicionados ahí, vamos a crear una serie de

carpetas que utilizaremos durante el proyecto.

La creación de éstas carpetas las puedes hacer desde el explorador de Windows o desde la consola. Nosotros utilizaremos la consola para crearlas, y utilizaremos para esto los comandos **dir** y **mkdir:**

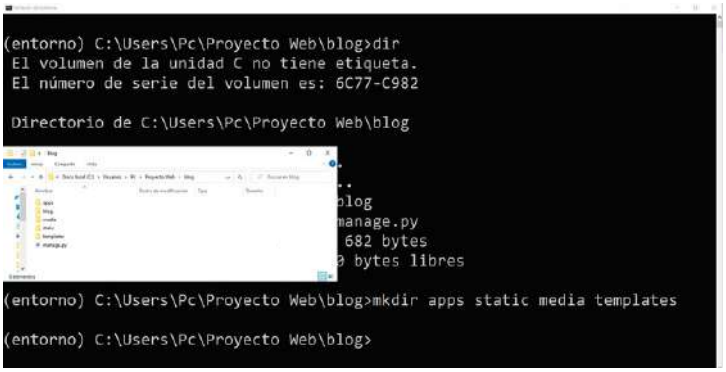
Con **dir** podemos verficar las carpetas y archivos que se encuentran dentro de la carpeta actual en la que estamos y con el comando **mkdir** vamos a crear las carpetas.

```
(entorno) C:\Users\Pc\Proyecto Web\blog>dir
 El volumen de la unidad C no tiene etiqueta.
 El número de serie del volumen es: 6C77-C982

 Directorio de C:\Users\Pc\Proyecto Web\blog

               <DIR>          .
               <DIR>          ..
               <DIR>          blog
                             682 manage.py
              1 archivos            682 bytes
              3 dirs  29.880.606.720 bytes libres

(entorno) C:\Users\Pc\Proyecto Web\blog>
```

ESTRUCTURANDO NUESTRO PROYECTO

PASOS

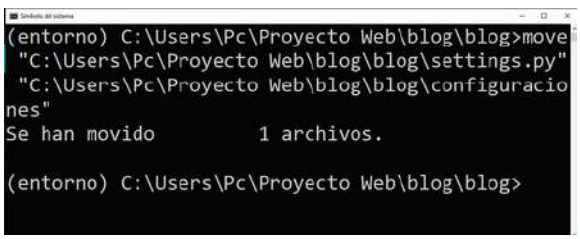
Vamos a crear una carpeta llamada "apps", luego una carpeta llamada "static", otra llamada "media" y otra "templates".



Puedes ver que con el comando **mkdir** y luego los nombres de las carpetas separadas por espacio, se crean todas las carpetas de una vez. En esta imagen puedes ver el comando y anexamos una captura de pantalla del explorador de Windows para que se pueda apreciar como se crearon las carpetas.
Ahora vamos a posicionarnos en la subcarpeta "blog" donde crearemos una carpeta llamada "configuraciones".

Si lo miramos desde el explorador de Windows podremos ver como a quedado nuestra estructura, o desde cmd, usando el comando **dir**.

Estando en esta carpeta vamos a mover el archivo settings.py a la carpeta configuraciones que acabamos de crear (puedes hacerlo desde la consola tal cual te mostraremos con el comando **move** o desde el explorador de Windows cortando y pegando el archivo a la carpeta "configuraciones").



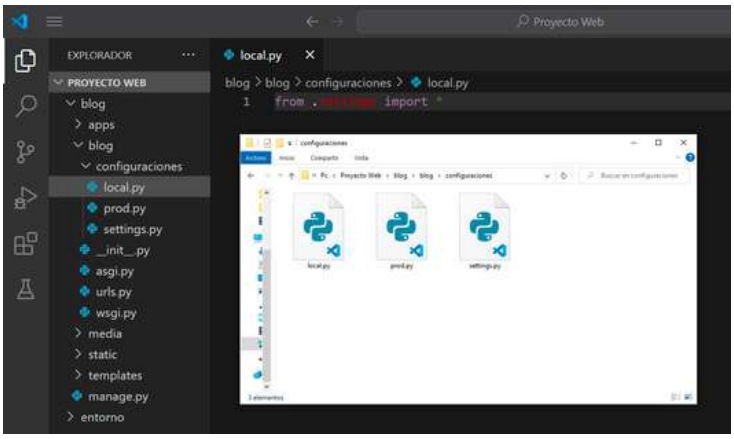
CREANDO LOS ARCHIVOS LOCAL.PY Y PROD.PY

>>> PASOS

Como puedes ver, usamos el comando **move**, luego la carpeta de origen y la carpeta de destino y, usamos comillas para poner la ruta de cada carpeta.

Hecha esta estructuración, podemos comenzar a trabajar con nuestro editor de código para realizar las primeras configuraciones y poner a funcionar nuestro servidor. Ya puedes cerrar la consola cmd que tienes abierta, ya que ahora trabajaremos con la consola cmd pero en VSC.

Abrimos Visual Studio Code y elegimos la carpeta de nuestro proyecto. Una vez ahí, vamos a ir a la carpeta de configuraciones y vamos a crear dos archivos nuevos: **local.py y prod.py** y, dentro del archivo **local.py** vamos a importar todo lo que haya en el archivo **settings.py**.



En esta captura de pantalla anexamos el explorador de Windows para que puedas ver como están creados los archivos (además de verlos en el explorador de VSC).

Luego abrimos el archivo **local.py** y en la primer línea de comandos escribimos **from .settings import *** para traer todas las configuraciones del archivo **settings.py** a nuestro archivo **local.py**.

Siempre que escribamos una línea de comando, debemos guardar el archivo, y una de las formas más rápidas es presionando la tecla **crtl + s**.

Ya verás, más adelante el motivo de la creación de estos archivos, pero por el momento, si estás siguiendo nuestros pasos, podrás seguir sin problemas.

Por útlimo, antes de correr nuestro servidor y verificar que nuestro proyecto está funcionando correctamente, vamos a realizar unas configuraciones más.

MODIFICANDO EL ARCHIVO SETTINGS.PY

>>> PASOS

Vamos a abrir el archivo **settings.py** para modificar ciertas puntos iniciales de modo tal que los siguientes pasos, sean más fluídos.

Vamos a guiarte por el número de línea que debes modificar dentro del archivo settings.py (no te preocupes que a medida que necesitemos, iremos explicándote las partes del código de este archivo) para que sea más fácil seguir los pasos ahora.

- En la línea de código 12, importaremos el módulo **os** que nos ayudará a interactuar con nuestro sistema operativo (gestionar archivos, directorios, variables de entorno, entre otras cosas).

```
12  import os
13  from pathlib import Path
```

- En la línea de código 57 vamos a configurar el acceso a nuestra carpeta "templates" (desde donde Django accederá a nuestras plantillas html para dar respuesta a las solicitudes HTTP).

```
56      'BACKEND': 'django.template.backends.django.DjangoTemplates',
57      'DIRS':[os.path.join(os.path.dirname(BASE_DIR),'templates')],
58      'APP_DIRS': True,
```

El código que estamos utilizando es el siguiente (para que puedas copiarlo específicamente):

'DIRS':[os.path.join(os.path.dirname(BASE_DIR),'templates')],

Lo que estamos haciendo con esta línea de código, es decirle a Django que una (join) la carpeta base (BASE_DIR) con la carpeta "templates" de forma que cuando se requiera acceder a un archivo html, vaya y los busque en esta carpeta "templates".

- En la línea de código 106 y 108 vamos a modificar el lenguaje que va a usar nuestro proyecto y la zona horaria:

```
106  LANGUAGE_CODE = 'es-ar'
107
108  TIME_ZONE = 'America/Argentina/Buenos_Aires'
109
```

En **LENGUAJE_CODE = 'es_ar'** para indicar que estamos en Argentina y el lenguaje será español y en **TIME_ZONE = 'America/Argentina/Buenos_Aires'** para indicar la zona horaria que vamos a utilizar.

- En la línea de código 119 vamos a decirle a Django de donde tiene que buscar los archivos estáticos que utilizará para nuestras plantillas html (como el logo de la página, botónes o imagenes que no cambiaran por todo el recorrido de la web):

MODIFICANDO EL ARCHIVO SETTINGS.PY

>>> PASOS

- El código que estamos utilizando es el siguiente (para que puedas copiarlo específicamente):

STATICFILES_DIRS = (os.path.join(os.path.dirname(BASE_DIR),'static'),)

- En la línea de código 126 y 127 definiremos el acceso a la carpeta media (la cual contendrá los archivos dinámicos que utilizarán nuestras plantillas, como puede ser una foto de perfil que suba un usuario que se registra):

- El código que estamos utilizando es el siguiente (para que puedas copiarlo específicamente):

MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(os.path.dirname(BASE_DIR),'media')

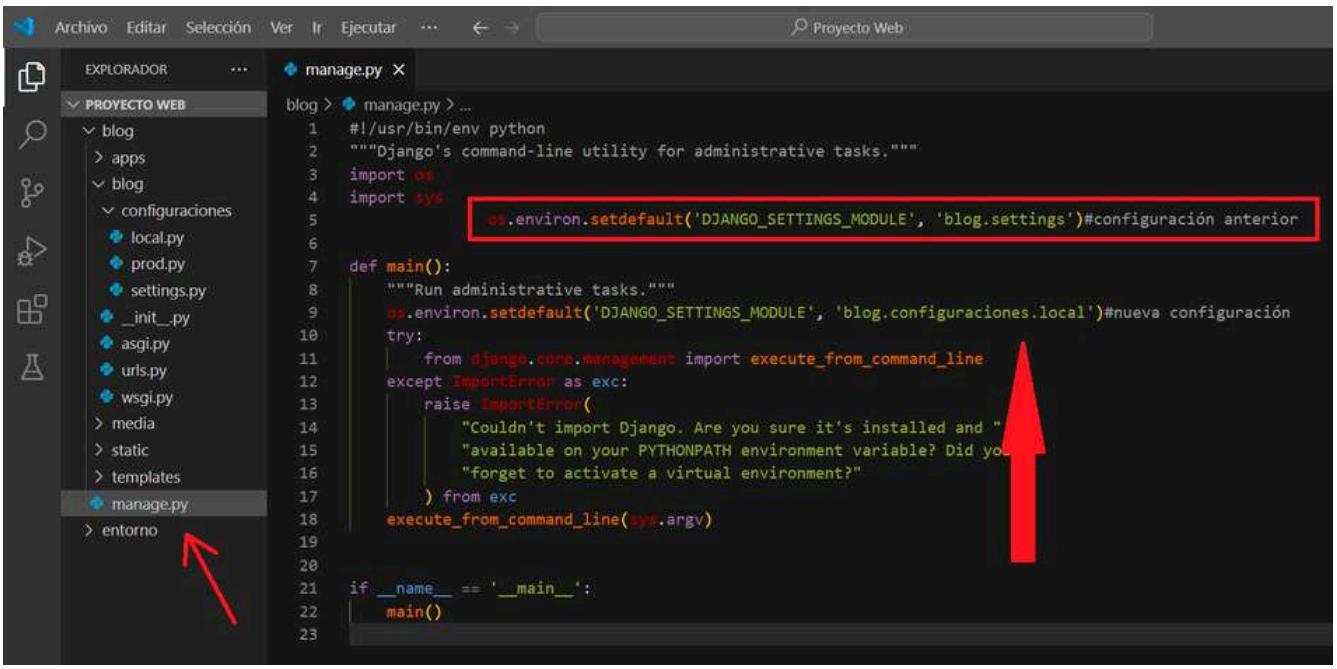
Al igual que hicimos con la carpeta "templates", en éstas configuraciones estamos declarando donde debe buscar Django los archivos estáticos que usaremos, como así también los archivos dinámicos o media que utilizaremos.

Con estas configuraciones iniciales ya podremos continuar con los siguientes pasos que serán correr nuestro servidor, abrir nuestra página html principal que indica que Django está corriendo nuestro proyecto y modificar nuestra plantilla o template html para la página de inicio sea una propia. Recuerda siempre guardar las modificaciones con **ctrl + s**.

CONFIGURANDO EL ARCHIVO MANAGE.PY

>>> PASOS

Para poder hechar a andar nuestro servidor, vamos a modificar el archivo **manage.py** (que se encuentra en nuestra carpeta principal):



Vamos a explicarte que hicimos aquí.

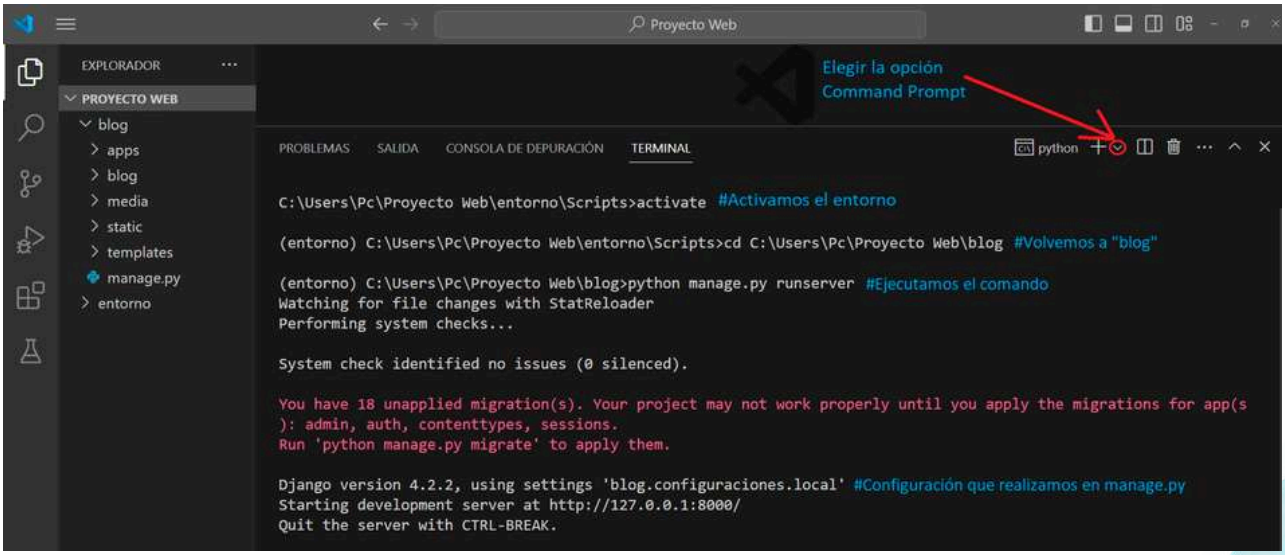
- Como lo mencionamos antes, el archivo **settings.py** tiene la configuración principal de nuestro proyecto, sin embargo, cuando trabajamos de forma local (en nuestra pc), tenemos configuraciones que solo vamos a usar en nuestra pc, pero cuando el proyecto se suba a nuestro repositorio y luego a un servidor, vamos a tener otras configuraciones. Para no estar reemplazando en el futuro las configuraciones solo en el archivo **settings.py**, dividimos las configuraciones dejando en l**ocal.py** las

EJECUTANDO EL SERVIDOR - RUNSERVER

configuraciones usadas al trabajar en nuestro entorno local y las configuraciones que vamos a usar en producción (cuando subimos el repositorio) las vamos a dejar en el archivo **prod.py**.
Lo que vemos en la captura de pantalla anterior es la modificación de la ruta desde la cual Django toma las configuraciones principales. Como creamos una carpeta llamada configuraciones y movimos el archivo settings.py, pero además, como en vez de settings.py vamos a usar el archivo **local.py** (para ciertas configuraciones), reemplazamos en la línea de código número 9 del archivo **manage.py** la ruta del archivo settings.py por la ruta al archivo **local.py** y te dejamos una captura de como se ve al principio la línea de comando número 9 y como se ve modificada quedando de esta manera:

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'blog.configuraciones.local')

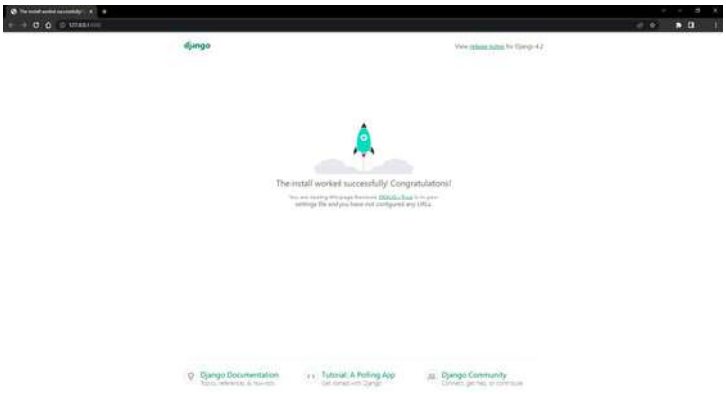
Terminada esta configuración (no olvides salvar cada modificación con ctrl + s) podemos poner a funcionar nuestro servidor para verificar que todo funciona correctamente y luego, cambiar la página inicial que nos brinda Django al momento de correr el servidor con nuestro proyecto.
Para poner a funcionar nuestro servidor, vamos a abrir la terminal (ctrl + ñ y elegir la opción Command Prompt) en VSC y luego, vamos a dirigirnos a la carpeta del entorno virtual para volver a activarlo y luego pondremos en marcha el servidor desde la carpeta "blog" donde se encuentra el archivo manage.py.



VIENDO LA PÁGINA DE INICIO DEL PROYECTO

>>> PASOS

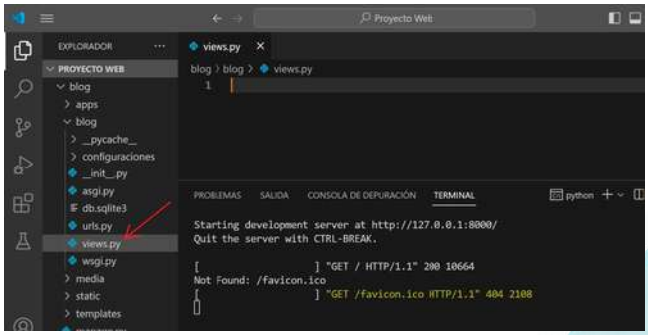
Una vez que el servidor se encuentra funcionando, podemos probarlo desde VSC, donde aparece la dirección web **http://127.0.0.1:8000/** (en la anteúltima línea de lo que se ejecutó), presionado la tecla **ctrl + click izquierdo** sobre esta dirección y se abrirá en nuestro navegador la pantalla de Django que nos informa que ya se ha creado el proyecto y se encuentra funcionando.
¡Felitaciones! si seguiste los pasos correctamente, ahora llegaste a esta pantalla:



Esta es la página principal, la cual vamos a modificar para que se pueda ver la página inicial que nosotros querámos en nuestro proyecto.
Por ejemplo: al momento de acceder a empleo.chaco.gob.ar vemos la página principal de la Subsecretaria de Empleo, la cual vendría a ser nuestra página index.html .

Ahora podríamos crear nuestra primer aplicación pero a fines de dar continuidad a esta página inicial, vamos a modificarla primero y luego crearemos las aplicaciones y cargaremos nuestros modelos.

Vamos a volver a VSC y crearemos un nuevo archivo llamado **views.py** en la carpeta "Proyecto Web/blog/blog"

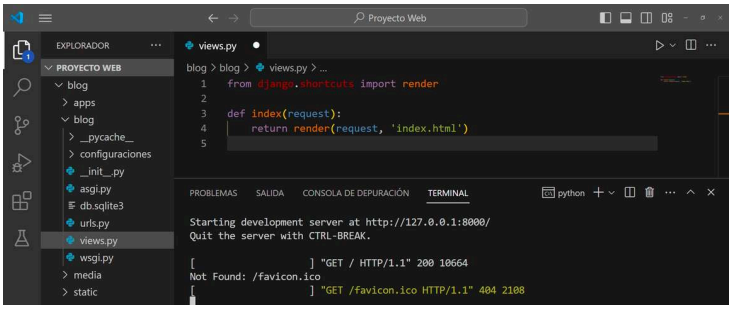


DECLARANDO VIEWS Y URLS

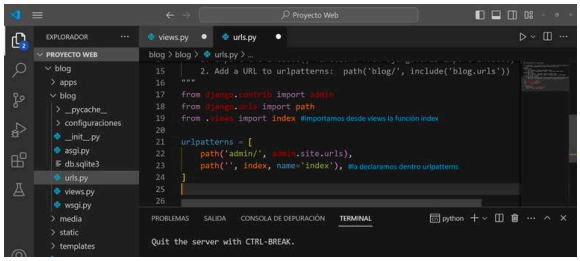
PASOS

El archivo **views.py** es un archivo importante en un proyecto Django que contiene las funciones o clases de vistas de tu proyecto. Las vistas son responsables de procesar las solicitudes entrantes y devolver las respuestas correspondientes (páginas web visibles por el usuario final).

Para poder mostrar esta página inicial, vamos a crear una función en nuestro archivo views.py, importando primero que nada una función que permitirá renderizar el archivo html que indiquemos:



Luego, vamos a ir al archivo urls.py y declararemos como se va a ejecutar este archivo html (que vamos a crear en el siguiente paso).



Como puedes ver, primero importamos desde views (va con un punto adelante para indicar el acceso -- **.**views) la función que creamos en el paso anterior "index", luego, debajo de el path "admin" declaramos como va a ser nuestro acceso desde la web. Fijate que en las 4 partes que tiene esta

DECLARANDO VIEWS Y URLS

>>> PASOS

declaración:

```
path('', index, name='index'),
```

- **path()**: sirve para indicarle a Django la ruta url que va a usar al momento de generar una respuesta a una solicitud HTTP.
- **''**: las comillas simples, sin nada adentro, significa que cuando se acceda a la página principal o inicial (como lo dijimos en el ejemplo de la página de oficina de empleo) va a mostrar el archivo html al que hacemos referencia (el que vamos a crear en el siguiente paso) que en este caso, va a ser el archivo index.html definido en nuestra **views.py**. Si por el contrario, quisieramos definir, supongamos la página de "contacto" (la cual veremos al avanzar más en el curso), deberemos colocar "contacto" dentro de éstas comillas simples.
- **index**: hace referencia al nombre de la función que creamos en el archivo **views.py** y que importamos en este archivo para poder usar.
- **name='index'**: nos servirá para hacer referencia a la página index, cuando incrustemos código Django en nuestros archivos html.

Puede parecer un poco confuso al principio, pero verás que una vez que comiences a practicar, podrás a comenzar a relacionar las partes y procesos a seguir para que todo funcione de forma armónica.

Antes de seguir con el siguiente paso, debemos resaltar que luego de cerrar el paréntesis, incluímos una coma (,) al final, la cual siempre tiene que ir al final de cada declaración de url ya que sin eso se puede generar un error al momento de visualizar el archivo html (tampoco olvides guardar cada cambio que estamos realizando con ctrl + s).

Ahora, el siguiente paso es crear nuestro archivo html a mostrar en respuesta a nuestra página principal.
Para ellos vamos a dirigirnos y nuestra carpeta **templates** y crearemos el archivo **index.html.**

Dentro de él, definiremos una estructura básica html (la cual, en este momento, los mentores dieron un taller para enseñar lo básico de html).

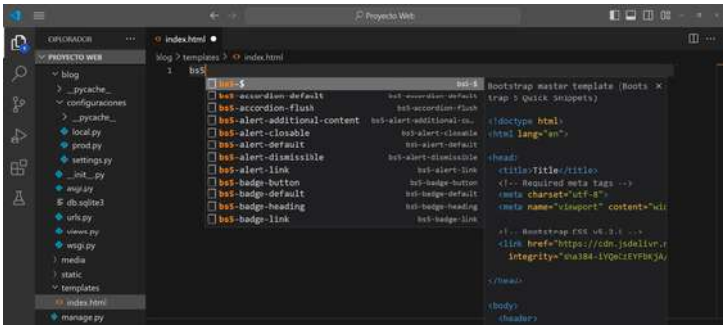
Ahora a fines prácticos cambiaremos el título de la página y el contenido que se verá dentro de la página.

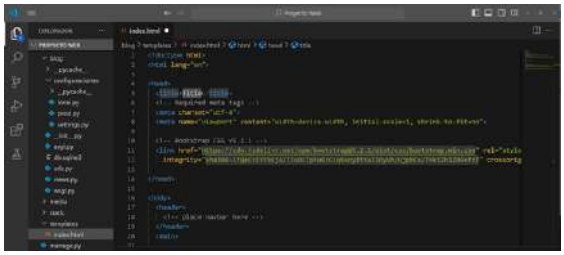
CAMBIANDO NUESTRA PÁGINA PRINCIPAL (INDEX)

PASOS

Antes de hacer la estructura básica html, te recomendamos descargar la siguiente extensión para VSC (no es necesario hacerlo, pero puede ayudarnos a escribir código html mucho más rápido):

vamos al archivo index.html que se encuentra en la carpeta "templates", escribimos bs5 y presionamos enter (si instalaste la extensión anterior) y nos saldrá de forma autocompletada la estructura básica de html:





En esta estructura básica cambiaremos el idioma en la línea de código 2, de inglés a español de argentina:

```
1  <!doctype html>
2  <html lang="es-ar">
3
```

En la línea de código 5, vamos a cambiar el nombre del título de la página de inicio:

```
4  <head>
5    <title>Proyecto final</title>
6    <!-- Required meta tags -->
```

Y en la línea de código 21 vamos a agregar una etiqueta <h1> para escribir un contenido que va a tener esta página de inicio:

```
20   <main>
21   <h1>Esta es nuestra página de inicio</h1>
22   </main>
```

Nos dirigimos al navegador web y actualizamos la página con la tecla f5 o presionando en el ícono de la flecha haciendo un círculo:

CAMBIANDO NUESTRA PÁGINA PRINCIPAL (INDEX)

Como puedes observar ya tenemos personalizada nuestra página inicial, principal o index, la cual va a ser nuestra página principal y desde ahí podremos navegar hacia las demás páginas.

Aclaramos que esta es solo una página muy básica de ejemplo, para mostrar como modificar y mostrar un documento html, a medida que avancemos, esto irá mejorando teniendo una web completa, visiblemente bien y funcional.

Básicamente vamos a utilizar la misma metodología para modificar el resto de la páginas y secciones pero con algunas sintaxis que debemos seguir para que Django pueda controlar bien toda la carga de código.

El hecho de hacerlo de esta forma, por un lado el código propio de Django y código de Python (que ya vamos a incrustarlo más adeltante), es para separar la parte lógica (backend) de la parte visual (frontend), para así poder separar responsabilidades y a la vez trabajar juntos desarrolladores backend y desarrolladores frontend sin inferir uno en el trabajo del otro. Además esto mejora la legibilidad del código para su mantención y hace que sea totalmente reutilizable (cuando crees otros proyectos, podrás tomar como base este y realizar algunas modificaciones para el nuevo proyecto sin tener que hacer todo desde cero) dando flexibilidad y adaptabilidad al cambiar o actualizar la apariencia visual de las páginas sin afectar la lógica subyacente.

A medida que vayamos avanzandando, iremos brindando más sintaxis propia del uso, pero ahora seguiremos con otro punto fundamental y de los más importantes que es la creación de nuestras aplicaciones.

CREANDO NUESTRA PRIMERA APP

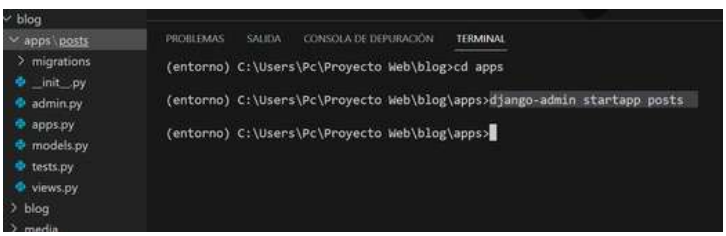
>>> PASOS

Este es de los apartados más importantes ya las aplicaciones serán nuestra columna vertebral para el funcionamiento de nuestro proyecto.

Hasta el momento vimos la interconexión entre la parte backend con el frontend, pero ahora comenzaremos a ver la modularización de Django lo cual lo hace uno de los frameworks más potentes y usados.

Comencemos:

- En primer lugar vamos a detener el servidor y esto lo hacemos presionando la tecla ctrl + c.
- Ahora vamos a movernos mediante el comando cd a la carpeta apps (si nos estás siguiendo, solamente debes escribir, luego de detener el servidor "cd apps").
- Una vez posicionados en esta carpeta, vamos a crear nuestra primer aplicación la cual, a modo de ejemplo, será "posts", y para ello utilizaremos el siguiente comando: **django-admin startapp posts**



Como puedes ver se ha creado una nueva carpeta llamada "posts" dentro de la carpeta "apps". Esta nueva carpeta "posts" es nuestra primer aplicación y dentro de ella podemos ver los archivos para que la misma funcione. Te explicamos brevemente cada uno:

- **__init__.py** (lo recordarás de cuando iniaciamos el proyecto): Este archivo vacío es necesario para que Python reconozca el directorio como un paquete. Puedes dejarlo en blanco, ya que Django se encarga de su funcionamiento interno.
- **admin.py**: Este archivo es utilizado para registrar los modelos de tu aplicación en el panel de administración de Django. Puedes personalizarlo para especificar cómo se muestra y se administra la información en el panel de administración.
- **apps.py**: Este archivo define la configuración de la aplicación. Puedes modificarlo para agregar metadatos adicionales o personalizar el comportamiento de la aplicación.

CREANDO NUESTRA PRIMERA APP

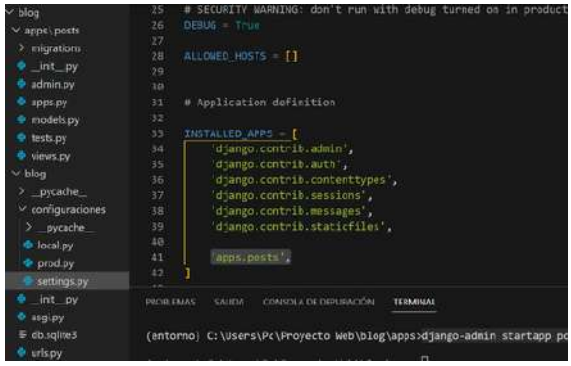
>>> PASOS

- **models.py**: En este archivo, defines los modelos de datos de tu aplicación utilizando la API de ORM de Django. Aquí especificas las clases que representan las tablas de la base de datos y sus campos.
- **tests.py**: Este archivo se utiliza para escribir pruebas automatizadas para tu aplicación. Puedes agregar casos de prueba para verificar el funcionamiento correcto de tus modelos, vistas y otras funcionalidades de la aplicación.
- **views.py** (el cual ya conoces): Aquí es donde defines las funciones o clases de vistas que manejarán las solicitudes HTTP y generarán las respuestas correspondientes. Puedes escribir lógica de negocio, interactuar con los modelos y renderizar plantillas HTML en las vistas.
- También se crea un directorio llamado "**migrations"**, que contiene archivos para gestionar las migraciones de la base de datos, pero estos no son archivos directamente visibles en el directorio de la aplicación.

Con la aplicación creada, la declararemos en nuestro archivo settings.py para que Django la pueda reconocer sin conflictos, por lo que vamos a abrir el archivo settings.py e iremos a la parte de "INSTALLED_APPS" (en la línea de código 33) y al final (línea 41), declararemos esta nueva aplicación de esta forma:

'apps.posts',

Con esto estamos dando la ruta de acceso que es la carpeta "apps" y, luego el nombre de la aplicación como tal. Esto lo hacemos entre comillas simples y al final si o si una coma (,). Por una cuestión de convención dejamos un espacion entre las aplicaciones pre-instaladas y las aplicaciones que crearemos.



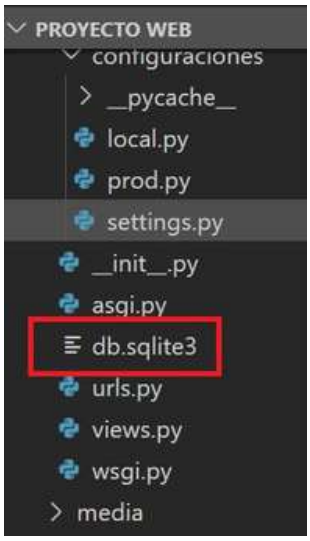
Además necesitaremos realizar otra configuración.
Debemos abrir el archivo **apps.py**, ubicado en carpeta apps/posts y modificar en la línea de código 6 la configuración de acceso agregando **'apps.posts'**

No olvides siempre guardar los cambios con ctrl +s.

CREANDO NUESTRA PRIMERA APP

>>> PASOS

Lo siguiente es realizar las migraciones.
Si has observado bien, te habrás dado cuenta que hay un archivo que no hemos explicado pero se creó al principio de nuestro proyecto. Estamos hablando del archivo "**db.sqlite3**".



Este archivo es una pequeña base de datos que Django nos brinda desde el inicio para poder realizar algunas pruebas, por lo que en este momento nos viene perfecto ya que la necesitaremos porque aún no hemos hecho la configuración de nuestra base de datos MySQL (la cual crearemos más adelante).

También, si eres buen observador, habrás visto que cuando corrimos por primera vez el servidor, nos aparecian unas letras rojas advirtiendo que no habíamos hecho las migraciones. Y esto se debía a que a modo de probar que el servidor corra, no era necesario, sin embargo, una vez que comenzamos a crear aplicaciones, vamos a requerir hacer las migraciones correspondientes para que se creen las tablas en nuestra base de datos o se realicen los cambios necesarios a medida que lo necesitemos (te adjuntamos la captura de pantalla de la advertencia de las migraciones para que lo recuerdes).

Para realizar las migraciones vamos a utilizar dos comandos: **makemigrations** y **migrate**. Estos son los comandos correspondientes para que se creen nuestras tablas y se carguen nuestros modelos (los cuales aún no hemos hecho, pero ya lo realizaremos).

MIGRACIONES

MAKEMIGRATIONS MIGRATE

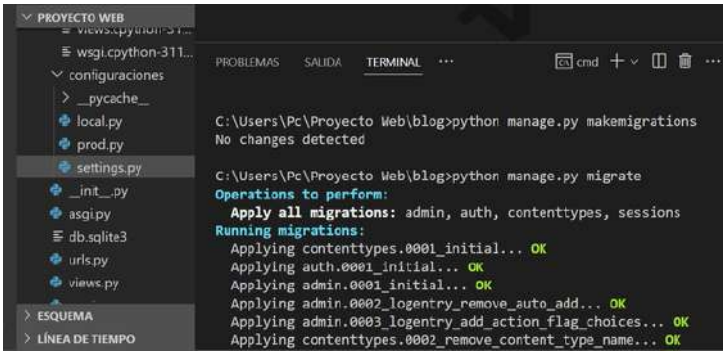
PASOS

En la consola vamos a posicionarnos en la carpeta "blog" con el comando **cd..**
Una vez posicionados ahí vamos a escribir el comando:

- **python manage.py makemigrations**

y luego el siguiente comando:

- **python manage.py migrate**



No te preocupes que al escribir **makemigrations** salga la leyenda "No chages detected", eso se debe a que aún no cargamos nada en nuestro modelo, pero te lo mostramos a modo de ejemplo para darte la recomendación que como buena práctica utilices los dos comandos juntos siempre, ya que alguna vez se nos puede pasar que hicimos un cambio en el modelo y luego si no utilizamos el comando makemigrations y solo utilizamos migrate, nos va a dar errores.
Con respecto a migrate, como puedes ver, se

comienzan a crear las tablas en nuestra base de datos (por el momento la base de datos sqlite3), pero vamos a aclarar que en este momento solo se están creando tablas que Django trae por defecto que son respecto a gestión de usuarios, las cuales, si tu profesor o profesora te dió un primer aceramiento al **admin** de Django, ya habrás podido ver de que se trata, pero si aún no lo has podido ver, no te preocupes que te lo mostraremos más adelante y tu profesor también lo hará.

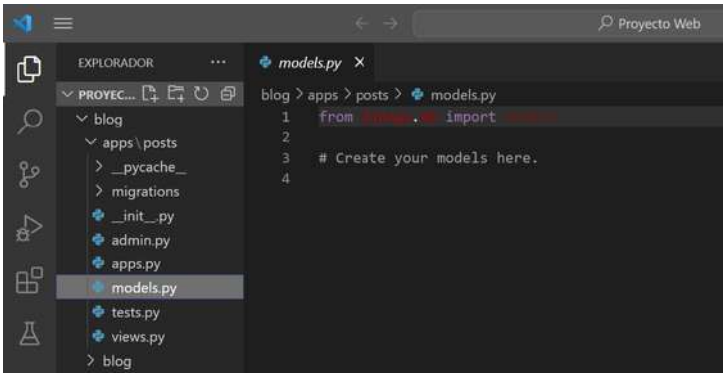
Para no hacer tan largo este apunte, lo dividiremos en varias partes y aquí haremos la primer división, aunque te adelantamos que, en la próxima parte crearemos el modelo de nuestra app "posts" y generaremos la vista correspondiente para que podamos visulizarla en nuestro navegador.

La recomendación en este punto es que realices este proceso (desde el inicio) desde la creación de un nuevo entorno virtual, pasando por un proyecto nuevo, la estructuración y configuración del mismo, así como la creación de una app, para que vayas familiarizandote mejor con todo este nuevo contenido.

CREANDO NUESTRO PRIMER MODELO

PASOS

Como pudimos adelantar en la primer parte del apunte, ahora vamos a cargar el modelo para nuestra aplicación "posts" y para eso tienes que abrir el archivo **models.py** que se encuentra en la carpeta "apps/posts" :



Te recordamos que los pasos que estamos siguiendo aquí (para todo) no son únicos, pueden haber diversas formas de crear un proyecto, apps, base de datos, pero siguiendo esta forma vas a poder crear tu modelo para el proyecto final sin problemas. Lo único que no va a cambiar, por más que la estructuración del proyecto no sea identica a esta, es la sintaxis que se requiere utilizar, por lo demás, puedes realizar variaciones.

Ahora si, manos a la obra:

- Vamos a arrancar en la línea de código 7 para dejar espacios y ser más organizados, donde primero

que nada vamos a hacer un comentario para mencionar que vamos a realizar la clase "Categoria":

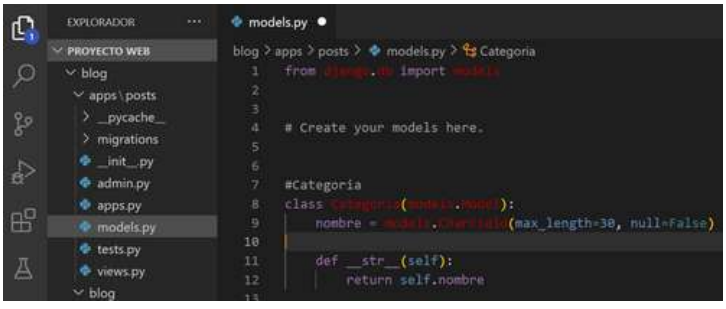
```
models.py
blog > apps > posts > models.py > Categoria
1   from django.db import models
2
3
4   # Create your models here.
5
6
7   #Categoria
```

Ahora crearemos una clase "Categoria" como lo hacíamos en POO, tomando a categoría como un objeto, y si lo vieramos desde SQL lo tomaríamos como una entidad.
Esta clase "Categoria" y, el resto de clases que crearemos para nuestros modelos, van a tener herencia de otra clase proporcionada por un módulo de Django - **django.db.models -** el cual se importa al iniciar (en la línea de código 1).
Al realizar esta herencia, estás indicando que esa clase es un modelo de datos que se puede almacenar y recuperar de una

CREANDO NUESTRO PRIMER MODELO

PASOS

base de datos utilizando la capa de abstracción de base de datos de Django.
Esto nos proporciona una serie de funcionalidades que nos brindan la POO y las bases de datos, como la definición de campos, métodos y atributos, relaciones entre modelos, abstracción de la base de datos ya que se obtiene acceso a una serie de características y funcionalidades útiles para trabajar con los modelos y la base de datos.



Puedes observar la herencia de la que estuvimos hablándote y, el formato que tiene la creación de la clase (como lo vimos en POO). En esta clase definimos el atributo "nombre" con un tipo de campo de tipo CharField que almacena strings con una longitud limitada (30 en este caso, ya que no se requeriría más de eso para un nombre de una categoría). Esto puede recordarte a como definiamos los campos en SQL, y si, todo está relacionado, por eso es que en este proyecto final estamos integrando todos los conocimientos adquiridos.

Además, separado por una coma, estamos indicando que el campo no puede ser nulo, por lo que se debe completar siempre con un nombre para la categoría.
Luego, más abajo, definimos el método "str" para cuando lo visualicemos, nos brinde el nombre de la categoría. Sin esta definición, cuando incluyamos una categoría, no nos mostraría el nombre, sino que aparecería como un "object" (te lo vamos a mostrar a modo de ejemplo más adelante).

Ahora nos tocaría definir la clase "Post" como tal, pero te contamos que definimos la clase "Categoría" primero para poder hacer una relación luego, entre las categorías y los posts. Ya que si no tenemos categorías, los posts no se podrían filtrar por categorías y, además, cada post quedaría desordenado. Si bien, se pueden aplicar filtros como "fecha de menor a mayor" por ejemplo, ya vas a notar lo necesario de definir una categoría para un post más adelante, y, sobre todo para este proyecto.
Antes de definir el modelo, importamos un módulo que vamos a utilizar:

CREANDO NUESTRO PRIMER MODELO

>>> PASOS

```
1   from django.db import models
2   from django.utils import timezone
```

El módulo "timezone" lo vamos a utilizar para que por cada post, nos guarde la hs actual de la creación.
Definimos el modelo y te explicamos las partes:

```
14  class Post(models.Model):
15      titulo = models.CharField(max_length=50, null=False)
16      subtitulo = models.CharField(max_length=100, null=True, blank=True)
17      fecha = models.DateTimeField(auto_now_add=True)
18      texto = models.TextField(null=False)
19      activo = models.BooleanField(default=True)
20      categoria = models.ForeignKey(Categoria, on_delete=models.SET_NULL, null=True, default='Sin categoría')
21      imagen = models.ImageField(null=True, blank=True, upload_to='media', default='static/post_default.png')
22      publicado = models.DateTimeField(default=timezone.now)
```

La primer parte ya te la explicamos con "Categoria" por lo que vamos a los atributos:

- **titulo**: va a ser de tipo CharField con una longitud de 50 (fuimos generosos), y además no puede ser un campo nulo ya que es el título del post.
- **subtitulo**: vendría a ser como un breve resumen que se puede escribir o no, por lo que "null" y "blank" los dejamos en "True", para que no sea estríctamente necesario que se complete.
- **fecha**: usamos el tipo DateTimeField diciéndole que tome la fecha y hora actual automáticamente cuando se cree un post.
- **texto**: para este campo que va a tener el contenido del post necesitamos no limitar la cantidad de strings que va contener, por lo que vamos a usar un tipo TextField, el cual no va a poder ser nulo y se podrá explayar todo lo necesario que requiera el contenido del post.
- **activo**: será para solo marcar una casilla (lo veremos más adelante), en la cual se tildará si un post está activo o no. Por defecto siempre que se crea un post estará activo (es decir, se publicará), sin embargo, este campo es realmente útil cuando solo queremos dejar de mostrar un post por un cierto tiempo sin tener que eliminarlo. Como puedes ver usamos un campo BooleanField que indica que este será un registro booleano.

CREANDO NUESTRO PRIMER MODELO

PASOS

- **categoria**: será nuestra clave foránea a la clase "Categoria", con lo cual podremos relacionar una categoria con un post, por ende para este campo usaremos el tipo ForeignKey indicando que la relación entre ambas clases (Categoria, Post). También hacemos una salvedad, en el parámetro on_delete. Lo que estamos haciendo ahí es indicar qué sucede cuando se elimina el objeto relacionado. En este caso, se establece en SET_NULL, lo que significa que si el objeto relacionado se elimina (es decir, si se elimina una Categoria), el campo "categoria" se establecerá como NULL (vacío). Además, no requerimos que un post se encierre si o si en una categoría al momento de crearlo, ya que puede pasar que sea crea un post y no hay una categoría específica para el tipo de post y no se la quiere crear en el momento, por lo que el campo "categoria" puede quedar vacio, sin embargo, lo que realmente vamos a hacer es incluirlo en una categoría "Sin categoría" por defecto. Esto nos facilitará luego, la recategorización de los post que queden sin categoría.
- **imagen**: este campo será de la parte dinámica de la web, por lo que si aún no te quedaba claro que significaba la carpeta media y la carpeta static, ahora podrás entenderlo mejor.

Las imagenes que pueda tener el post, se guardarán mediante este campo gracias al tipo

ImageField que está preparado para aceptar archivos de tipo imagen. Este campo como puedes ver, tiene parámetros que indican que puede quedar sin completarse (null y blank = True), sin embargo, si es que ocurre esto, pondremos una imagen que estará por defecto (default='static/post_default.png') y así nos aseguraremos que todo post contenga una imagen. Las imagenes que se carguen cuando se cree un post, se subirán a la carpeta "media" ya que éstas imagenes forman la parte dinámica de la web, donde el o los usuarios que realicen la carga de post, irán subiendo o modificando las imagenes conforme se requiera, pero para las imagenes que permaneceran sin modificarse estarán en la carpeta "static" y, con esto nos referimos archivos como el logo de la web, el archivo de imagen para un post que no tendría imagen o un archivo de imagen para un usuario que se registra y no pone una imagen de perfil, entre otros archivos más que estarán indicados como fijos en la web (esto no quiere decir que los archivos

CREANDO NUESTRO PRIMER MODELO

>>> PASOS

dentro de la carpeta "static" no pueden ser modificados, solo quiere decir que van a ser archivos que en primera instancia, permanecerán ahí sin ser modificados o cambiados).

- **publicado**: aquí también utilizamos un tipo DateTimeField como en el campo "fecha", pero cambia el parámetro que le indicamos aquí. Te mostraremos mejor la diferencia entre el parámetro de "fecha" y este parámetro cuando hagamos una publicación.

Ahora definiremos una clase interna para nuestro modelo mediante la clase "Meta":

```
24      class Meta:
25          ordering = ('-publicado',)
```

Meta es una clase interna dentro de la definición del modelo que define los metadatos para ese modelo en particular.

Se utiliza para definir metadatos adicionales sobre un modelo. Uno de los metadatos más comunes que se puede definir en la clase "Meta" es "ordering", que especifica el orden en que los objetos del modelo deben ser consultados o recuperados de la base de datos.

Además, como hicimos con la clase "Categoria", definiremos el método "__str__" y un método más que nos servirá para que cuando eliminemos un post, también se eliminen las imagenes asociadas a ese post. De esta forma no saturaremos el servidor con imagenes de post que ya no existen.
Este método se definirá con "delete" que realizará dos acciones adicionales antes de eliminar el post. Primero, elimina el archivo asociado al campo imagen y luego realiza la eliminación estándar del objeto llamando al método delete() de la clase padre. Esto permite añadir un comportamiento personalizado al proceso de eliminación del objeto.

CREANDO NUESTRO PRIMER MODELO

>>> PASOS

```
24      class Meta:
25          ordering = ('-publicado',)
26
27      def __str__(self):
28          return self.titulo
29
30      def delete(self, using = None, keep_parents = False):
31          self.imagen.delete(self.imagen.name)
32          super().delete()
```

Adicional al método "delete" usaremos los parámetros opcionales "using" y "keep_parents" para contolar el comportamiento específico durante la eliminación del objeto del modelo.

- **using** = None: El parámetro using se utiliza para especificar la base de datos en la cual se realizará la eliminación del objeto. Por defecto, se establece en None, lo que indica que se utilizará la base de datos predeterminada definida en la configuración de Django. Si deseas eliminar el objeto de una base de datos específica distinta a la predeterminada, puedes pasar el nombre de la base de datos como argumento al llamar al método delete().
- **keep_parents** = False: El parámetro keep_parents se utiliza en situaciones donde el modelo tiene herencia de tablas (herencia de modelos). Si se establece en True, se mantendrán los registros en las tablas de los modelos padre después de eliminar el objeto actual. Sin embargo, en este caso específico, se establece en False, lo que significa que los registros de los modelos padre no se mantendrán y se eliminarán junto con el objeto actual.

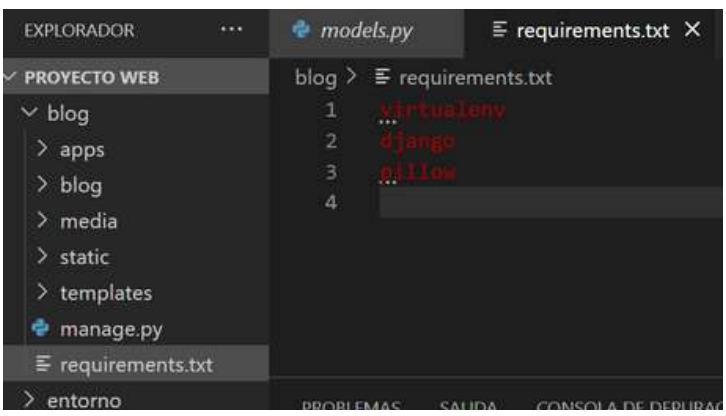
Una vez cargado nuestro modelo y guardado (ctrl + s), iremos al siguiente paso.

Como usaremos imagenes en nuestro modelo, necesitamos instalar una dependencia más aparte de las que tenemos instaladas que son "virtualenv" y "django". Esta nueva dependencia será "pillow" quien se encargará de gestionar nuestras imágenes.
A medida que sigamos avanzando vamos a requerir instalar más dependencias, por lo que es un buen momento para crear un archivo fundamental que llevará el registro de las dependencias que estamos utilizando en nuestro proyecto. Este es el archivo "**requirements.txt**".
Este es un archivo de texto donde iremos anotando las dependencias con las que trabajamos en nuestro proyecto, de manera tal que, si necesitamos mudarnos de pc, por ejemplo, o si subimos nuestro

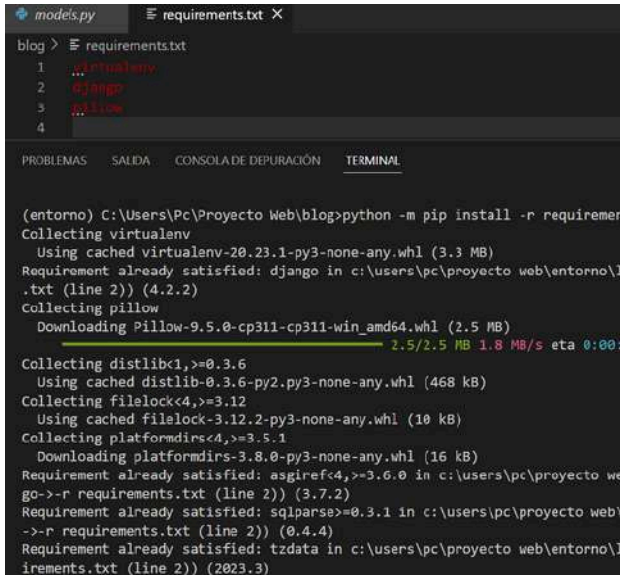
INSTALANDO DEPENDENCIAS ADICIONALES - REQUIREMENTS.TXT

>>> PASOS

repositorio a Github y alguien quisiera descargarlo y trabajar con él, con solo un comando podría instalar todas las dependencias usadas en el proyecto.
Este archivo lo podremos crear de forma manual o automática. Ahora lo haremos de forma manual, pero luego te daremos el comando para hacerlo de forma automática.
Vamos a posicionarnos en la carpeta "blog", donde se encuentra el archivo "manage.py", ahora crearemos un archivo llamado "requirements.txt" (puedes hacerlo desde VSC, desde el explorador de Windows o desde el cmd).
Una vez creado este archivo vamos a escribir las dependencias que tenemos instaladas y la que vamos a instalar, la cual, como mencionamos antes, será "pillow" y guardamos el archivo (ctrl + s).



Ahora, en la consola ejecutaremos el comando "**python -m pip install -r requirements.txt**".



Puedes observar que se vuelve a instalar la dependencia "virtualenv", "django" y ahora "pillow".
Este archivo "requirements.txt", puede ser instalado al principio del proyecto, no necesariamente tiene que hacerse en este

INSTALANDO DEPENDENCIAS ADICIONALES - REQUIREMENTS.TXT

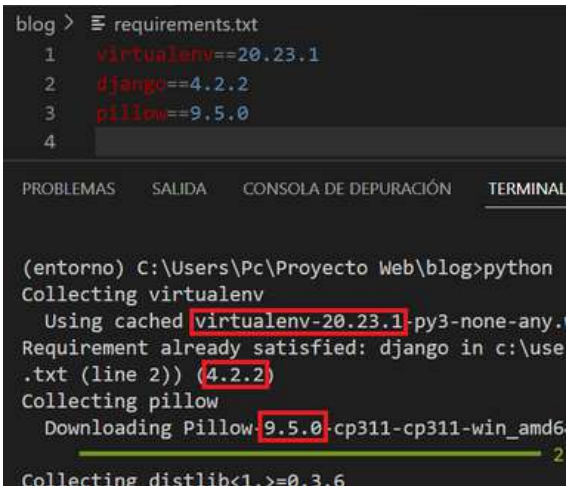
>>> PASOS

punto, pero lo hicimos de esta manera para seguir una continuidad.

Hacemos hincapié en que la estructuración del proyecto y los pasos seguidos hasta aquí no necesariamente, tienen que ser igual para cada proyecto, hay convenciones, pero no necesariamente tiene que ser así, por lo que ahora lo hicimos de esta forma para que puedas visualizar de una vez, las versiones que tenemos instaladas de nuestras dependencias, la cuales, vamos a dejarlas anotadas en nuestro archivo "requeriments.txt" ya que si por algún motivo, en una nueva actualización de estas dependendencias quitan o agregan algo, nuestro proyecto puede llegar a tener errores o incluso no funcionar. Además, anotamos la versión para no usar versiones anteriores o posteriores, ya que también pueden ocacionar inconvenientes en nuestro proyecto (generalmente, irregularidades en el funcionamiento).

Ahora vamos a mirar cada versión de éstas 3 dependencias que tenemos instaladas y las dejaremos anotadas en nuestro archivo.

Para dejar correctamente la versión de cada dependencia usamos la asignación con doble signo igual (==) seguida de la versión que estamos utilizando actualmente.



Con nuestro archivo "requirements.txt" completo, hasta este momento, ya podemos realizar nuestras migraciones para poder crear las nuevas tablas del modelo que acabamos de cargar.

Lo último que queda, antes de realizar las migraciones, es mostrarte el comando para que el archivo "requeriments.txt", se cree de forma automática.

INSTALANDO DEPENDENCIAS ADICIONALES - REQUIREMENTS.TXT

>>> PASOS

Para realizarlo, debemos ejecutar en la consola parte de un comando que aún no te mostramos y es un buen momento de hacerlo.

Este comando lo usamos para verificar todas las dependencias que tenemos instaladas. Este comando en particular es "**pip freeze**" y, como te adelantamos nos muestra las dependencias instaladas en nuestro entorno:

```
blog > requirements.txt
1   virtualenv==20.23.1
2   django==4.2.2
3   pillow==9.5.0

PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL

(entorno) C:\Users\Pc\Proyecto Web\blog>pip freeze
asgiref==3.7.2
distlib==0.3.6
Django==4.2.2
filelock==3.12.2
Pillow==9.5.0
platformdirs==3.8.0
sqlparse==0.4.4
tzdata==2023.3
virtualenv==20.23.1

(entorno) C:\Users\Pc\Proyecto Web\blog>
```

Como puede observar, este comando nos arroja las depencias instaladas en nuestro entorno y sus versiones. Te darás cuenta que no solo están las dependencias que instalamos manualmente, sino que hay otras más. Las otras dependencias que no instalamos manualmente, fueron instaladas automáticamente con las dependencias que nosotros instalamos.

Para hacer la creación automática del archivo "requirements.txt" vamos a utilizar el comando:

- **pip freeze > requirements.txt**

Se ejecutas este comando verás que el archivo "requirements.txt" generado de forma automática, instala todas las dependencias que existen en nuestro entorno ahora, las que fueron de instalación manual, como las que fueron de instalación automática, por lo que al finalizar el proyecto y antes de subir el repositorio final a Github, te recomendamos ejecutar este comando nuevamente para que nada se pase por alto.

EJECUTANDO LAS MIGRACIONES PARA NUESTRO MODELO

PASOS

Ahora es momento de generar las migraciones de nuestro modelo creado y para ello vamos a seguir los pasos que realizamos anteriormente.
Ejecutaremos el comando **python manage.py makemigrations** y el comando **python manage.py migrate**

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL

(entorno) C:\Users\Pc\Proyecto Web\blog>python manage.py makemigrations
No changes detected

(entorno) C:\Users\Pc\Proyecto Web\blog>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, posts, sessions
Running migrations:
  No migrations to apply.

(entorno) C:\Users\Pc\Proyecto Web\blog>
```

Si bien nos aparece el mensaje "No changes detected" al realizar "makemigrations", lo hacemos, como te mencionamos antes, por una cuestión de usar estos comandos juntos. Aún sale este mensaje porque si bien, generamos un nuevo modelo, no realizamos ningún cambio sobre él, sin embargo con "migrate" si se han creado las nuevas tablas en nuestra base de datos.
Cuando comencemos a realizar ciertos cambios en nuestras tablas, "migrations" mostrará el mensaje correspondiente.

Como puede observar, al ejecutar "migrate" se ha

creado la tabla "Post" y esto es por la carga del modelo que hicimos.

Para poder ver nuestro modelo ya funcionando y poder comenzar a hacer algunas cargas de datos de prueba, vamos a hacer uso de una herramienta fundamental que nos brinda Django y es su administrador.

Si miras en el archivo "urls.py" (donde cargamos nuestra página de inicio), recordarás que pusimos nuestra url debajo de otra ya existente, y esa url es la que nos dirige al administrador que nos ofrece Django.

urls.py

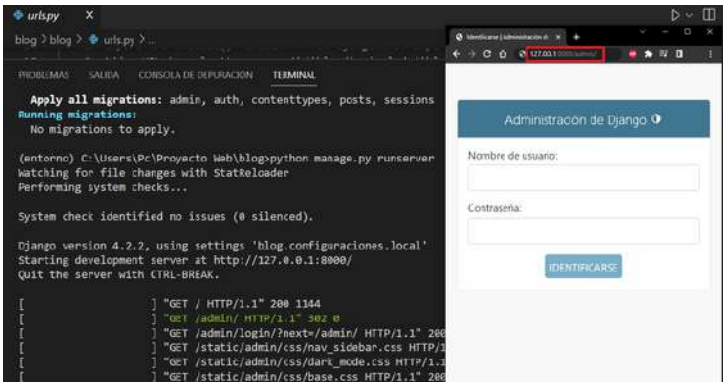
blog > blog > urls.py > ...

```
15      2. Add a URL to urlpatterns:  path(
16  """
17  from django.contrib import admin
18  from django.urls import path
19  from .views import index
20
21  urlpatterns = [
22      path('admin/', admin.site.urls),
23      path('', index, name='index'),
24  ]
```

ACCEDIENDO AL ADMINISTRADOR DE DJANGO

>>> PASOS

Pues bien, si de momento quisieramos acceder a esta página, lo podriamos hacer, pero no tendríamos acceso al uso de ella, ya que nos requiere un usuario y contraseña.
Ejecutamos nuestro servidor y lo probamos:



Para acceder solo debemos poner en la barra de direcciones, luego del localhost o la dirección ip del servidor "/admin" e ingresarás a la página (**http://127.0.0.1:8000/admin/**).
Sin embargo, por más que intentes poner cualquier usuario y contraseña, no podrás acceder, y esto es porque aún no hemos generado nuestro usuario y contraseña para el administrador de Django.

Para hacerlo, vamos a detener el servidor y ejecutaremos el comando:

- **python manage.py createsuperuser**

Inmediatamente nos aparecerá un mensaje en el que nos pedirá que ingresemos un nombre de usuario:

```
(entorno) C:\Users\Pc\Proyecto Web\blog>python manage.py createsuperuser
Nombre de usuario (leave blank to use 'pc'):
```

Como estamos trabajando en un entorno local, y por una convención usaremos el nombre "root" y lo mismo será para los siguientes mensajes de contraseña y repetir contraseña (usaremos "root" para todo), y por último, le diremos que "si (yes - y)" cuando nos pregunte si estamos seguros (debido a lo insegura de la contraseña).

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

(entorno) C:\Users\Pc\Proyecto Web\blog>python manage.py createsuperuser
Nombre de usuario (leave blank to use 'pc'): root
Dirección de email: root@root.com
Password:
Password (again):
La contraseña es muy similar a nombre de usuario.
La contraseña es demasiado corta. Debe contener por lo menos 8 caracteres.
La contraseña tiene un valor demasiado común.
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.

(entorno) C:\Users\Pc\Proyecto Web\blog>
```

notarás que en email pusimos un mail que no existe, solo es de prueba, sin embargo, es requerido que se complete con formato de email (algo@algo.com)

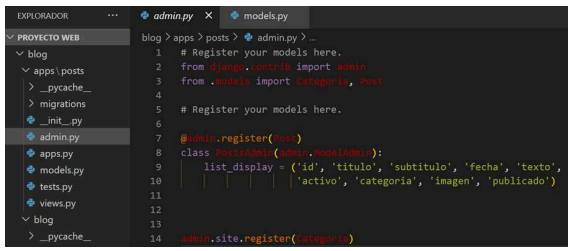
ACCEDIENDO AL ADMINISTRADOR DE DJANGO

PASOS

Hecho esto y con el mensaje que nos dice Django que el Superusuario ha sido creado, ejecutamos el servidor y recargamos la página "admin" donde quedamos. Ahí completaremos con "root" para usuario y contraseña e ingresaremos.

Dentro del sitio tendremos este menú, donde explicaremos cada cosa en la siguiente división del material.
Ahora, lo que podemos observar es que el modelo que cargamos "Post" no aparece y eso es porque nos falta un paso más para que pueda aparecer aquí, y este paso lo tendremos que hacer con todas las aplicaciones que crearemos más adelante, para que puedan visulizarse desde aquí.

Vamos a dirigirnos al archivo admin.py ubicado en la carpeta apps.
Dentro de este archivo, lo completaremos de la siguente forma (no olvides que siempre debes guardar los cambios con crtl + s):



Te mostramos las líneas a completar con zoom:

```
1   # Register your models here.
2   from django.contrib import admin
3   from .models import Categoria, Post
```

```
7   @admin.register(Post)
8   class PostsAdmin(admin.ModelAdmin):
9       list_display = ('id', 'titulo', 'subtitulo', 'fecha', 'texto',
10                      'activo', 'categoria', 'imagen', 'publicado')
```

```
14  admin.site.register(Categoria)
```

Vamos a explicar las partes:

- En las líneas de código 2 y 3 estamos realizando las importaciones necesarias para que funcione todo correctamente. Por un lado hacemos una importación propia del admin de Django y por otro lado, traemos desde el archivo models.py las clases que creamos.

ACCEDIENDO AL ADMINISTRADOR DE DJANGO

PASOS

- Luego en la línea de código 7 usamos la sintaxis para hacer el registro de la clase "Post".
- En la línea 8 declararemos una clase propia de Django.
- Para luego poder usar un "list_display" con los atributos que queramos mostrar (enseguida lo vas a ver) propios de la clase "Post".
- Por último, en la línea 13, hacemos un registro simple de la clase "Categoría".

Hecho esto, salvamos (crtl + s) y nos dirigimos al navegador, donde actualizamos la página.

Y ahora si ya podemos ver nuestras clases del modelo que creamos.
Este sistema administrador que nos brinda Django, es una ayuda inmensa ya que podemos gestionar la carga de nuestra base de datos con una interfaz gráfica

funcional. Esto nos facilita mucho ya que es mucho más intuitivo que hacerlo mediante código en nuestra base de datos, nos ahorra tiempo ya que no tuvimos que crear esta interfaz, es decir, no tuvimos que crear un template para comenzar a gestionar, con lo cual, ya podemos comenzar a hacer pruebas e ir corrigiendo algunas cosas que aún no están afinadas. Incluso, hay dependencias que permiten modificar todo este sitio, pudiéndolo dejar de forma muy personalizada y con esto incluso, no es tan necesario crear un sitio completo para la administración de los datos para el usuario o cliente final.

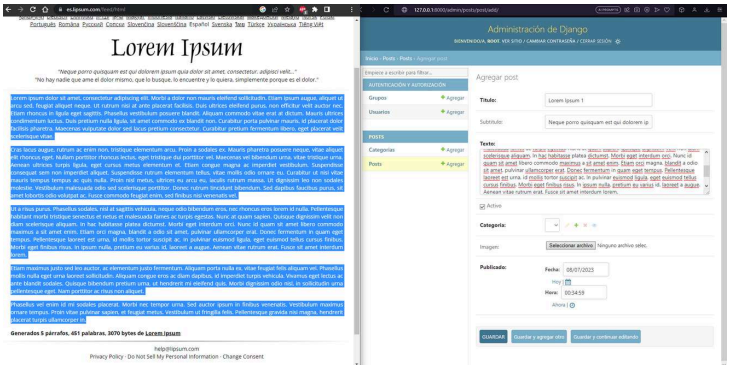
Hasta aquí llegaremos con esta parte del apunte. En la próxima parte comenzaremos a hacer pruebas y cambiaremos los templates.

COMENZANDO A HACER PRUEBAS

PASOS

Como puedes observar, iremos copiando y pegando para crear contenido.
Para la sección de Categorías, haremos click en el símbolo + para añadir una categoría y crearemos una nueva.

Luego seleccionaremos una imagen de prueba y crearemos nuestro primer artículo con el botón "Guardar".

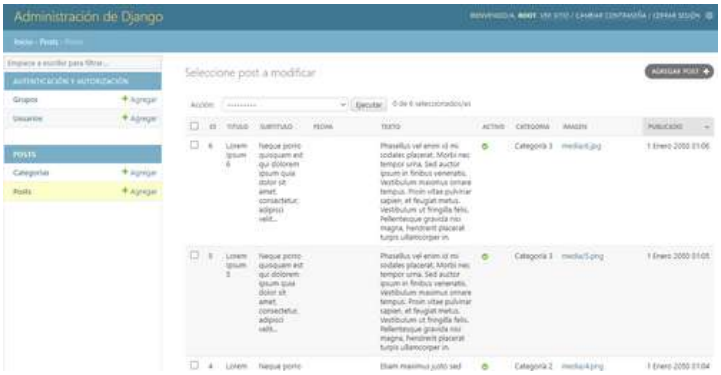


Verás una pantalla como esta donde saldrá un mensaje en la parte superior, avisandonós que se guardó correctamente.

COMENZANDO A HACER PRUEBAS

>>> PASOS

Vamos a reducir el texto del artículo y crearemos 5 más para las pruebas.



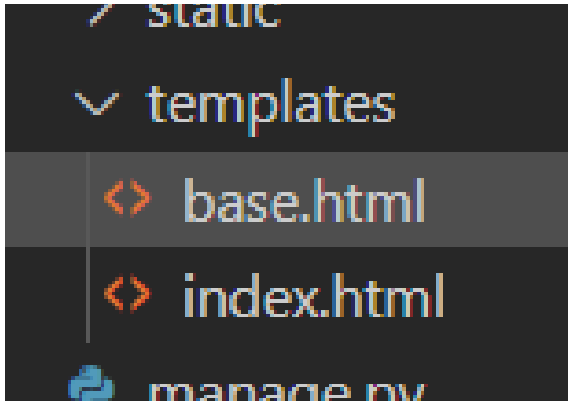
Como puedes observar cada artículo tiene su "id", así como el "título", "subtítulo", "texto", y todas las etiquetas que definimos para que se mostrarán en el "list_display" que definimos previamente en admin.py

```
@admin.register(Post)
class PostsAdmin(admin.ModelAdmin):
    list_display = ('id', 'titulo', 'subtitulo', 'fecha', 'texto',
                    'activo', 'categoria', 'imagen', 'publicado')
```

Ahora es momento de mostrar estos artículos mediante html, pero primero que nada, vamos a hacer unas modificaciones en el html principal, y para eso vamos a crear un nuevo html que va a contener todas las cosas que el resto de los html también contendrá. Y esto te sonará de algún lado... es porque vamos a usar herencia, pero en este caso se va a heredar de un

html a otro html.

Vamos a crear un nuevo archivo html dentro de nuestra carpeta templates con el nombre de base.html.



Dentro de este html usaremos (como lo vimos en páginas anteriores) Bootstrap 5 para generar una estructura html (escribimos bs5, presionamos enter y tendremos nuestra estructura).

MODIFICANDO HTMLS

Ahora que ya cargamos la estructura base, vamos a comenzar a cargar la etiqueta **{% load static %}** en la primer línea de nuestro html, que es una etiqueta personalizada de Django que permite cargar los archivos estáticos configurados previamente en settings.py.
Luego vamos a cambiar el lenguaje a "es-ar" para que sea español de Argentina.
Más abajo, vamos a cambiar la etiqueta "title" para con la etiqueta de Django **{% block title %} Página de inicio {% endblock title %}** donde lo que hacemos es que mediante esta etiqueta en cada html que se acceda el título que se ve en la pestaña de la página pueda ir cambiando ("Página de inicio" es solo un nombre de referencia, puedes poner el nombre que quieras).

Además vamos a crear un navbar genérico con Bootstrap 5 y vamos a comenzar a armar.
Además como para ir organizando y que comience a verse mejor, vamos a agregar una imagen de cabecera. Esto lo vamos a hacer con la etiqueta **{% static 'img/info.jpg' %}**, donde "img" será la subcarpeta dentro de la carpeta "static", donde vamos a comenzar a poner todos los archivos estáticos, es decir los archivos que no van a cambiar a lo largo del uso de la web. Estos van a ser la imagen de como para la "marca" de la página, archivos css o js que usaremos, etc. En este caso estamos haciendo referencia a la ruta de la carpeta static --> img ---> y el nombre del archivo (en este caso info) con su extensión (.jpg).

MODIFICANDO HTMLS

Y si lo vemos en el html (con la estructura para un navbar genérico), quedaría algo así.

```
<!-- Required meta tags -->
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

<!-- Bootstrap CSS v5.2.1 -->
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.1/dist/css/bootstrap.min.css" rel="stylesheet"
  integrity="sha384-iYQeCzEYFbKjA/T2uDLTpkwGzCiq6soy8tYaI1GyVh/UjpbCx/TYkiZhlZB6+fzT" crossorigin="anonymous">

</head>

<body>

  <header>
    <a class="flex-sm-fill text-sm-center nav-link active" href="#">
        <img src="{% static 'img/info.jpg' %}" style="max-width: 300px; margin: 30px;">
    </a>

    <nav class="nav nav-pills flex-column flex-sm-row">
        <a class="flex-sm-fill text-sm-center nav-link active" href="#">Active</a>
        <a class="flex-sm-fill text-sm-center nav-link" href="#">Longer nav link</a>
        <a class="flex-sm-fill text-sm-center nav-link" href="#">Link</a>
        <a class="flex-sm-fill text-sm-center nav-link disabled" href="#" tabindex="-1" aria-disabled="true">Disabled</a>
    </nav>
```

Lo siguiente es configurar el "bloque de contenido", que será la parte del html que irá cambiando según la solicitud http que se pida. Es decir, si vamos a la página de inicio, se mostraría el mismo navbar que en todas las páginas que recorramos, pero cambiaría el contenido principal. Este bloque de contenido lo declararemos en la parte media del html, entre el navbar y el footer (luego haremos el footer), pero recuerda que esto es solo una estructura guía. Cada uno puede incluir la estructura que desee, pero no te preocupes si aún no entiendes muy bien esta parte, ya se irán aclarando las dudas a medidas que sigamos avanzando.
El bloque de contenido lo declararemos usando la etiqueta **block:**

MODIFICANDO HTMLS

```
25      <nav class="nav nav-pills flex-column flex-sm-row"
26          <a class="flex-sm-fill text-sm-center nav-link
27          <a class="flex-sm-fill text-sm-center nav-link
28          <a class="flex-sm-fill text-sm-center nav-link
29          <a class="flex-sm-fill text-sm-center nav-link
30      </nav>
31
32      {% block contenido %}
33
34      {% endblock %}
35
36
37    <footer>
38      <!-- place footer here -->
39    </footer>
```

Puedes ver que la sintaxis utilizada es

{% block contenido %}
{% endblock %}

donde esta será la parte dinámica de la web.
Crearemos el footer ahora y luego comenzaremos a mostrar lo que estamos haciendo.
Lo pondremos entre las etiquetas **<footer> </footer>**, tal cual lo hicimos con el navbar con sus etiquetas **<nav> </nav>**.

```
36
37    <footer id="sticky-footer" class="flex-shrink-0 py-4 bg-dark text-white-50">
38      <div class="container text-center">
39        <small>Copyright &copy; Your Website</small>
40      </div>
41    </footer>
42
```

Declaramos un Footer genérico.

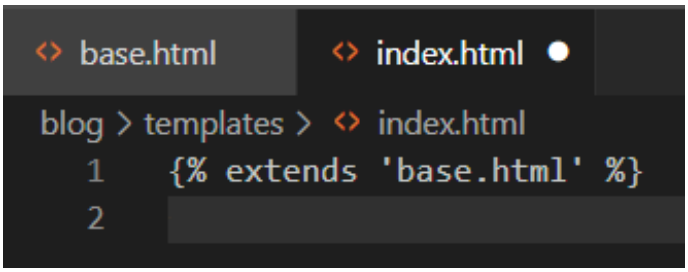
```
39        <small>Copyright &copy; Your Website</small>
40      </div>
41    </footer>
42
43
44  <!-- Bootstrap JavaScript Libraries -->
45  <script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.6/dist/umd/popper.min.js"
46    integrity="sha384-oBqDVmMz9ATKxIep9tiCxS/Z9fNfEXiDAYTujMAeBAsjFuCZSmKbSSUnQlmh/jp3" crossorigin="anonymous"
47  </script>
48
49  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.1/dist/js/bootstrap.min.js"
50    integrity="sha384-7VPbUDkoPSGFnVtYi0QogXtr74QeVeeIs99Qfg5YCF+TidwNdjvaKZX19NZ/e6oz" crossorigin="anonymous"
51  </script>
52</body>
53
54</html>
```

Por último y para finalizar la estructura simple de html que tenemos, van unas líneas de código donde se declaran scripts JS que trae por defecto.

Es momento de realizar la herencia de este template base a otros templates. Arrancamos con index.html

Si recuerdas en la página 16 y 17 de este apunte, realizamos nuestro primer html llamado "index" en el que mostrábamos el mensaje "Esta es nuestra página de inicio". Además si recuerdas, realizamos una estructura básica para este html, sin embargo, en "base.html" también acabamos de realizar una estructura base con componentes como el Navbar y el Footer que van a ser partes estáticas en todos los html que crearemos. Así que ahora es momento de realizar la herencia de templates o plantillas.
En index.html vamos a borrar todo el código que hay dentro y vamos a usar la siguiente etiqueta de Django para hacer la herencia desde base.html a index.html

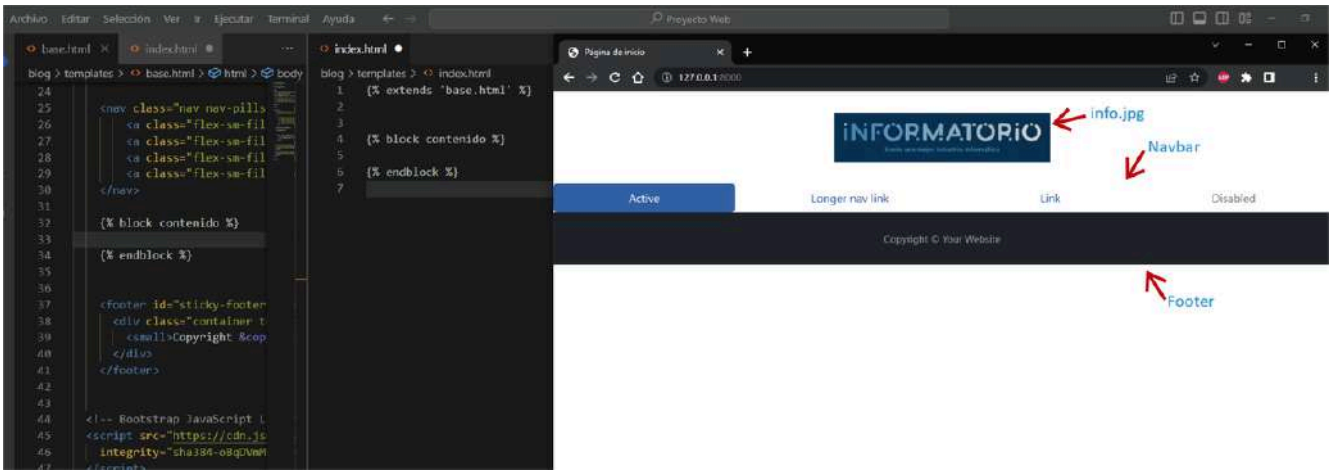
MODIFICANDO HTMLS



Puedes ver que la sintaxis utilizada es

{% extends 'base.html' %}

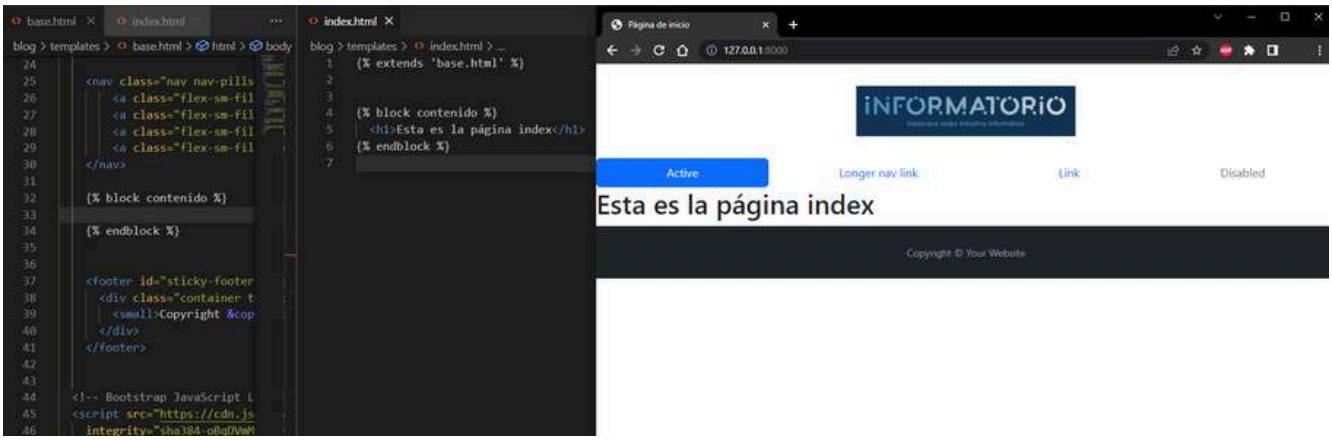
Con esto estamos diciendo a Django que tiene que heredarle a index (o también se podría decir como "extenderle a index") todas las declaraciones en base.html que van a estar fuera de **{% block contenido %}** y **{% endblock %}** de esta forma, llamando desde index.html a "block contenido" vamos a poder cargar todo lo que queremos mostrar en el cuerpo de index.html
Te lo mostramos solamente llamando a "block contenido" sin agregar nada y luego agregando algo para que puedas ver la diferencia y lo comprendas mejor:



De lado izquierdo dividimos la consola para que puedas, ver a la vez, el archivo base.html y el archivo index.html, y del lado derecho pusimos la página index para que veás como está armada. Se ve la imagen que pusimos, más abajo el Navbar con los botones por defecto y el Footer. En medio de éstos dos iría la parte del "block contenido", pero como no hay nada en este momento, en esa parte, en

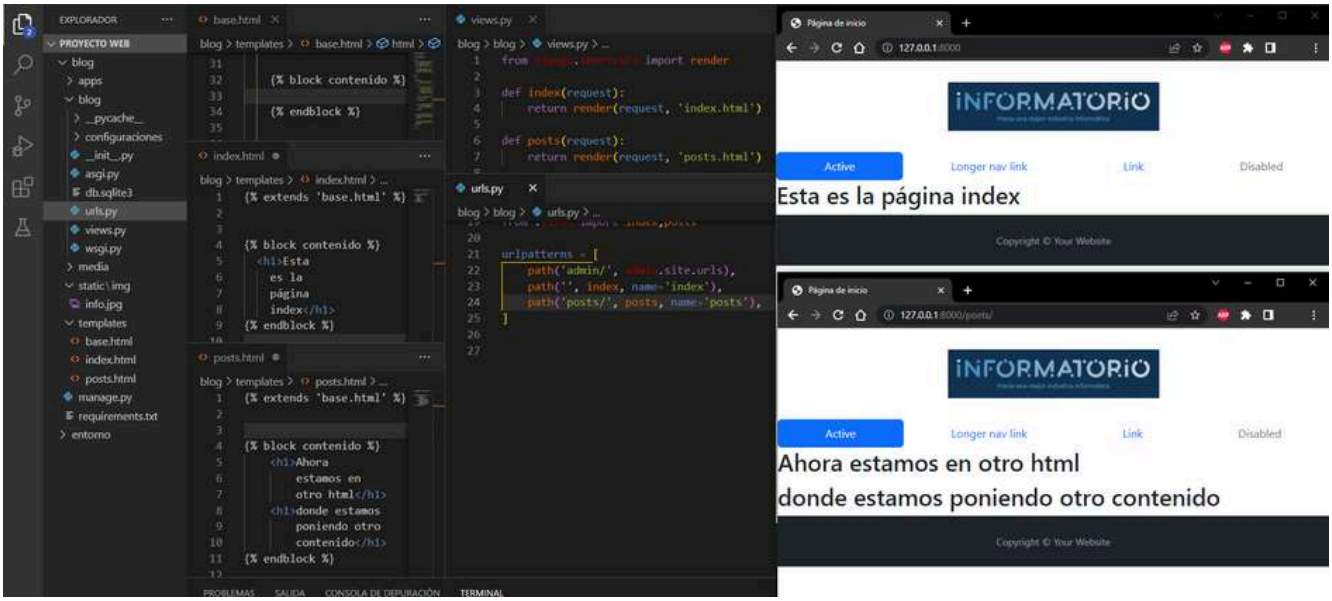
MODIFICANDO HTMLS

index.html, solo se muestra lo que viene por herencia de base.html ¿Pero que pasa si agregamos en index.html?



Puedes ver que el contenido que agreguemos en index.html se ve reflejado cuando actualizamos la web. Esto es porque todo lo que se encuentre dentro del "block contenido" que declaramos en base.html pero lo usamos por medio de herencia en index.html, se verá reflejado, mientras que lo que esté fuera del "block contenido" seguirá de la misma forma en todos los html que hereden de base.html.
Para dejarlo más claro, vamos a hacer un nuevo html llamado posts.html donde haremos una herencia, pero pondremos otro mensaje en la parte dinámica (block contenido) para reflejar el cambio de html.
Lo vemos en la siguiente página con el código, además de la creación de la views y url para el otro html que usaremos.

DECLARANDO VIEWS, URLS Y CONTENIDO DE HTML



Parece una pantalla complicada, pero la hicimos así para que sea visualmente lo más completo posible. Te explicamos cada cosa de izquierda a derecha:

- Lo primero que hicimos fue crear otro archivo html: **posts.html**
- Dentro de posts.html hicimos lo mismo que con index.html, pero cambiamos el contenido dentro del "bloque contenido".
- Después creamos una vista en views.py para poder mostrar el html creado.
- Por último definimos en urls.py la forma de acceso que iba a tener el html --> **127.0.0.1/posts**

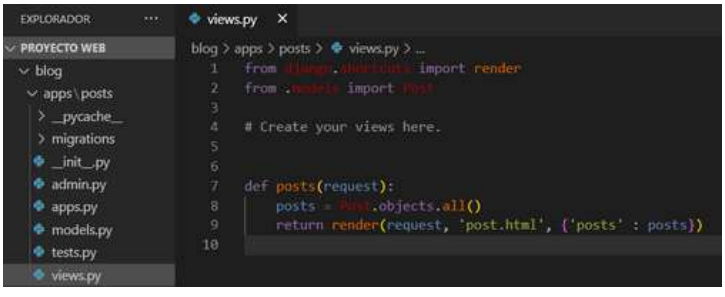
Puedes observar que la imagen superior, el Navbar y el Footer no cambian, pero si la parte del "block contenido". De esta manera podemos hacer toda la herencia que necesitemos, para la cantidad de htmls que necesitemos, sin tener que escribir todo el código inicial de la estructura del html. Esta es una de las características más potentes de Django y la forma que nos deja reutilizar el código.

Ahora podemos comenzar a hacer que se vea un poco mejor visualmente toda la web, pero antes te vamos a mostrar como introducir código Python en Django para poder mostrar los posts como un listado según la cantidad de posts cargados que tengamos en nuestra base de datos.

DECLARANDO VIEWS, URLS Y CONTENIDO DE HTML

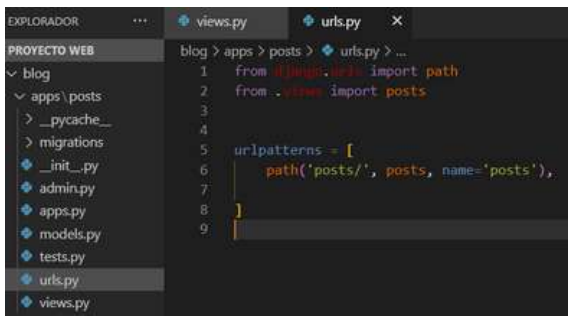
PASOS

Primero que nada vamos a crear la vista respectiva de "posts" desde la views.py de la aplicación "posts". Como viste antes, la creamos, a modo de ejemplo, desde la views principal, pero ahora vamos a comenzar a ordenar, por lo que crearemos las vistas respectivas desde cada aplicación. Por el momento solo tenemos la aplicación "posts", pero más adelante crearemos otras.
Ahora para mostrar los posts cargados en nuestra base de datos vamos a dirigirnos a views.py de "posts" y crearemos una vista basada en funciones en primera instancia, para mostrarte los tipos de vista que se pueden usar (esta vista es la que venimos viendo hasta el momento):



Lo primero que hicimos fue importar desde "models" el modelo "Post". Luego, definimos la vista basada en función, donde dimos un contexto (articulos), que lo que hace es tomar todos los elementos que se encuentran en "Post" y los trae hacia la vista, que en el

"return", aparte de pasarle el html que tiene que mostrar, le pasamos otro parámetro más que es el contexto: **{'posts' : posts}**
Con esto ya estamos extrayendo los registros que tenemos cargados en la base de datos para "Post" en la variable "posts".
Lo siguiente es crear la url, pero en nuestra aplicación "posts", sin embargo, este archivo no existe, por lo que vamos a crearlo:

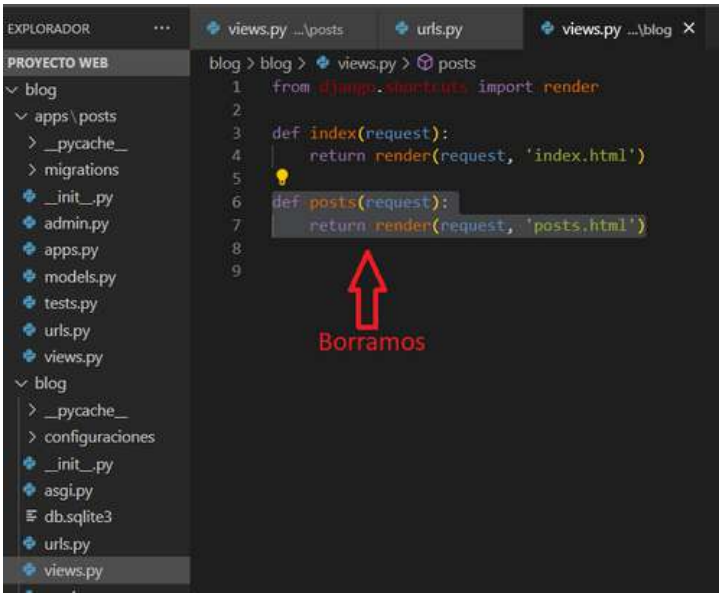


Donde dentro de este archivo urls.py vamos a importar el "path" desde "django.urls" (tal cual en las urls.py principales) y luego traemos la vista "posts" que creamos en views.py

DECLARANDO VIEWS, URLS

PASOS

Guardamos los cambios, nos dirigimos a la views.py principal (blog/views.py) y borramos la vista "posts" que tenemos creada ahí ya que la que vamos a utilizar ahora es la de la aplicación.

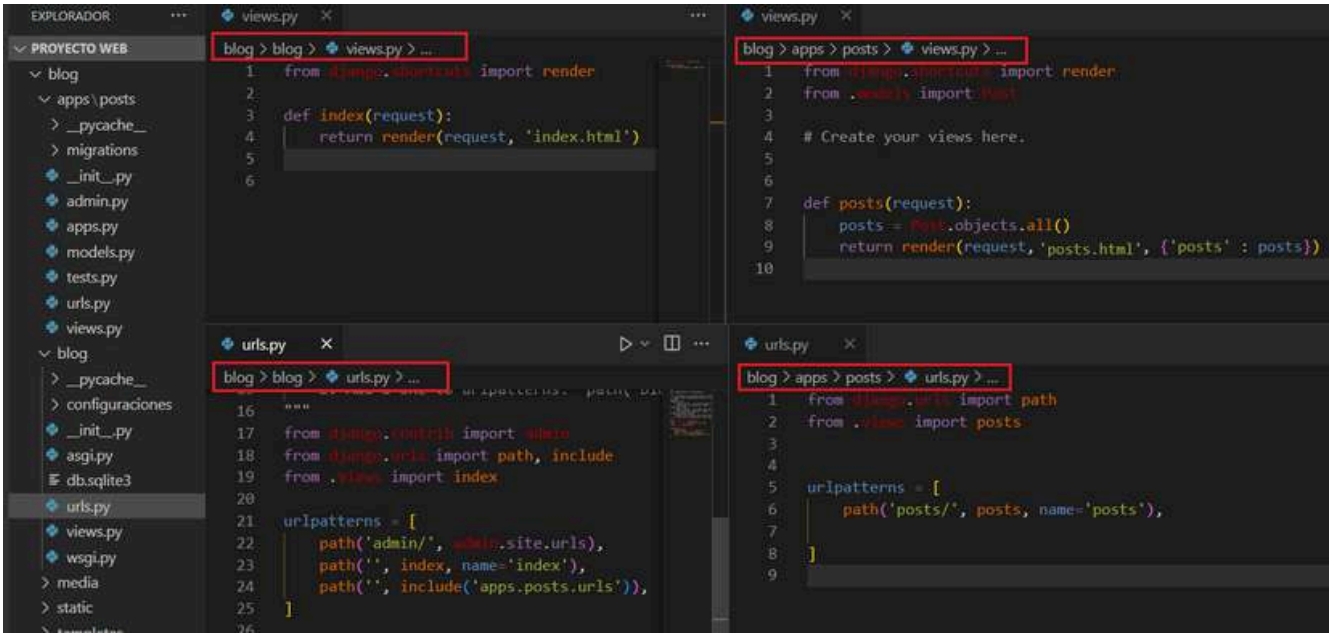


Guardamos los cambios y nos dirigimos a el archivo urls.py principal (blog/urls.py) para hacer una modificación ya que desde este archivo, ahora solo tenemos que "avisar" que vamos a utilizar la url de la aplicación "posts".

```
from django.contrib import admin
from django.urls import path, include
from .views import index

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', index, name='index'),
    path('', include('apps.posts.urls')),
]
```

En este urls.py principal, lo que vamos a hacer es importar "**include**" luego de "path" desde django.urls y vamos a declarar mediante "include" que vamos a utilizar la urls de la aplicación "posts".
Este es el camino que vamos a seguir siempre, ya que las vistas y urls que utilicen nuestras aplicaciones, las crearemos desde las aplicaciones, pero, siempre va a ser necesario, mediante "include", declarar que tenemos que usar la url que se encuentra en nuestra aplicación.
A continuación te mostramos la pantalla dividida de los que son las views.py tanto de la aplicación "posts" como las principales, lo mismo para las urls.
Con esto puedes controlar que todo te haya quedado igual.

DECLARANDO VIEWS, URLS E INTRODUCIENDO CÓDIGO PYTHON



Para no perderte, siempre puedes mirar abajo de la pestaña que está abierta, la ruta de donde se encuetra el archivo abierto. Te lo marcamos en rojo para que lo puedas ver bien.
Una vez hecho todo esto, guardamos todo (nunca debemos olvidarlo, siempre, siempre guardar cada cambio) y ya podemos modificar el html "posts" para que se muestre la lista de registros en nuestra base de datos.

Luego en la imagen más pequeña te dejamos el detalle de la modificación y el uso de Python dentro del código html.
Como puedes ver, utilizamos un "for" tal cual lo usabamos en Python, con una variable que va a hacer la iteración (i) en "posts" que es la variable donde se guardan todos los registros de nuestra base de datos, la cual pasamos como contexto en nuestro html "posts".
La sintaxis es especifica de Django con las llaves de apertura y cierre y los signos porcentuales.

posts.html

blog > templates > posts.html > ...

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

{% block contenido %}


    {% for i in posts %}

        <br>
        <li>{{ i.titulo }}</li>
        <li>{{ i.subtitulo }}</li>
        <li>{{ i.categoria }}</li>

        {% empty %}
        <h1>No hay registros</h1>

    {% endfor %}

    <br>
    <h1>Ahora
        estamos en
        otro html</h1>
    <h1>donde estamos
        poniendo otro
        contenido</h1>

{% endblock %}
```

AGREGANDO CÓDIGO PYTHON EN DJANGO

>>> PASOS

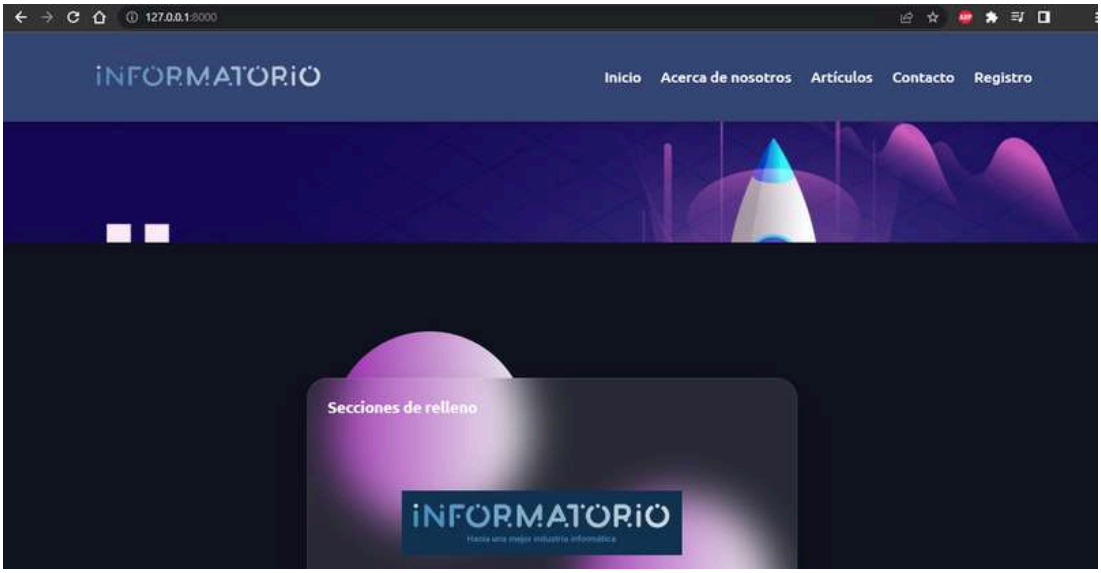
Luego utilizamos una sentencia más que es "empty" con la misma sintaxis de "for" que lo que hace "empty" es servir de cláusula para que en el caso de que no hayan registros en la base de datos, muestre el texto "No hay posts", ya que si por algún motivo, la base de datos no tuviese ningún registro y no tenemos esta cláusula, la página se mostraría vacía, y eso es un completo error.
Para finalizar usamos "**endfor"** con la misma sintaxis requerida para Django.
Además has observado que para llamar a los campos de cada registro, utilizamos una sintaxis con doble apertura y cierre de llave, la variable de iteración, luego un punto y seguido, el campo al que estamos llamando.
Podemos llamar a todos los campos que necesitemos, pero a modo de ejemplo, aquí solo llamamos a los campos, "título", "subtitulo" y "categoria".
Las etiquetas
, <li>, son propias de html.

```
{% for i in posts %}

    <br>
    <li>{{ i.titulo }}</li>
    <li>{{ i.subtitulo }}</li>
    <li>{{ i.categoria }}</li>

    {% empty %}
    <h1>No hay registros</h1>

{% endfor %}
```

En este momento, ya podemos hacer algunas modificaciones en nuestros htmls para que se puedan ver mejor, es decir la parte del frontend, ya que básicamente estuvimos trabajando sobre el backend.

Vamos a hacer modificiones, sin embargo, esto no necesariamente es parte del curso, ya que se está dando en paralelo en las clases de mentoría todo lo que es html, css y js, por lo que solo vamos mostrarte las capturas de pantalla de como quedaría con algunas modificaciones para que se vea más bonito nuestro blog. Usaremos Bootstrap 5 para generar líneas de código html, crearemos un archivo css para estilos y un archivo javascript para dar algunas animaciones.
Esto solo es de ejemplo ya que al finalizar el curso, cada grupo presentará distintas formas de presentación aprendidas en clases, además se podrán descargar plantillas html prediseñadas para dar una mejor visual del trabajo.
A continuación te mostramos nuestras modificaciones de ejemplo para que puedas animarte a hacerlo también.

ANIMÁNDOTE A QUE PUEDAS MODIFICAR EL FRONTEND DE TUS HTMLS



ANIMÁNDOTE A QUE PUEDAS MODIFICAR EL FRONTEND DE TUS HTMLS

Aquí cerraremos esta parte. En la próxima te mostraremos lo que sería la vista de los "posts" con el html modificado, además te enseñaremos a incluir imagenes, ya que esto lleva más sintaxis y declaraciones ({% load static %} , {{ posts.imagen.url }}) entre otras cosas más. Además comenzaremos a crear otras aplicaciones.

Para que puedas seguir cada paso que vamos dando, es muy importante que respetes las sintaxis y no dejes pasar nada. Revisa varias veces si te da algún error al momento de ejecutar, porque de seguro solo son errores de sintaxis.

Este apunte se va realizando a medida que se va probando, por lo que si te surge algún error, es porque pusiste algo demás o de menos. Verifica bien cada cosa.

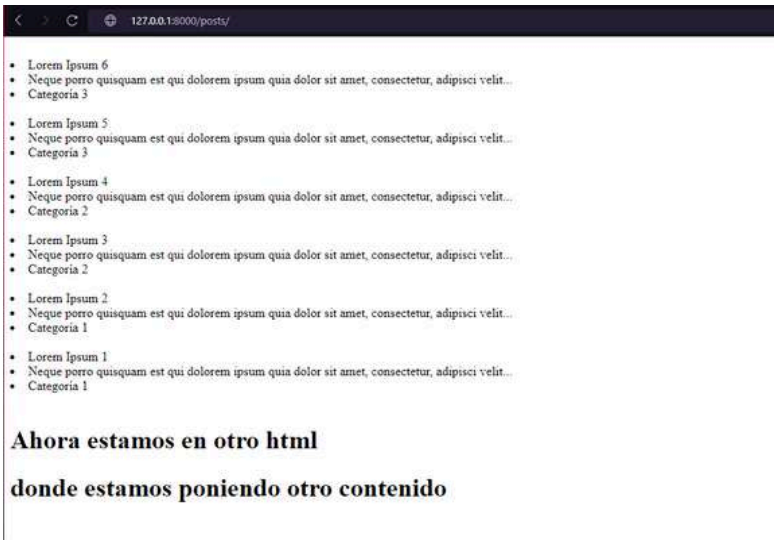
Hay cosas que pueden variar, pero otras que no, por lo que te recomendamos que hasta tener más práctica, sigas las mismas sintaxis que este apunte, así puedes seguir tal cual lo vas viendo aquí.
Además en las clases podrás incorporar otras cosas para usarlas en tu proyecto.

MOSTRANDO EL HTML "POST"

>>> PASOS

Como te mencionamos en la parte anterior, vamos a mostrar "posts.html" desde el navegador. Vamos a usar el código que tenemos cargado y eso será suficiente para mostrar lo que hay en la base de datos.

Si lo pudiste ver, nosotros ya tenemos modificados nuestros html base con css y js para que tenga una mejor visual, sin embargo, esto no afectaría a lo que realmente tienes que hacer como trabajo backend.
También te mencionamos que para hacer estas modificaciones e incluir css y js, utilizamos Bootstrap, tanto desde la extensión que instalamos en páginas anteriores, como desde la página de Bootstrap (https://getbootstrap.com/) la cual indica la instalación en el html base, haciendo declaraciones al principio del html. Puedes leer los pasos y documentación de esta página para lograr acabados similares o mejores a la página de prueba que te estamos mostrando.
Aclarado esto, vamos a "post.html" y antes de heredar de "base.html" como ya lo hicimos con "index", te vamos a mostrar como se vería la página sin hacer la herencia (ya que tenemos la vista creada y el url) accedediendo a ella mediante http://127.0.0.1:8000/posts/



MODIFICANDO NUESTRA VISTA BASADA EN FUNCIONES POR UNA DE CLASES

Te dejamos la captura de "posts.html" (realizado en la página 45 y 46) para que puedas ver que con el código Python colocado dentro de la plantilla html podemos visualizar (accediendo mediante la url creada) los registros "posts" de nuestra base de datos (los que creamos con anterioridad desde el admin de Django).

Se puede observar también, que no tiene ningún formateo ni nada ya que no estamos herendando de base.html ni aplicando css ni js. y, por el momento, lo vamos a dejar así y al terminar de explicarte esta parte, aplicaremos la parte visual para que veas como se vería. Ahora, a modo de ejemplo y para que entiendas sin tanto código html de por medio, lo dejamos así.

Lo siguiente a esto, sería poder acceder a cada post de forma individual, para poder tener el contenido completo del texto, además, más adelante cuando creemos nuestros comentarios, será desde esta entrada individual al "post" que podremos generar a estos comentarios.

Para poder acceder a un post de manera individual, debemos hacer referencia al "id" del "post", y esto lo tenemos que hacer desde la url, previa creación de la view para esto, por lo que vamos a ir a nuestra view de la aplicación para extraer el id del post y poder ingresar a él.

Sin embargo, vamos a enlazar un tema más que son las **vistas basadas en Clases**.

Ya que tenemos nuestra vista para "posts" creada en base a funciones, ahora vamos a hacer la misma, para en base a Clases y lo hacemos de la siguiente manera:

apps > posts > views.py > ...

```python
from django.shortcuts import render
from .models import Post
from django.views.generic import ListView

#Vista basada en funciones
def posts(request):
    posts = Post.objects.all()
    return render(request, 'posts.html', {'posts': posts})

#Vista basada en clases
class PostListView(ListView):
    model = Post
    template_name = "posts.html"
    context_object_name = "posts"
```

Estas vistas hacen lo mismo (en este caso), muestran el contenido de la misma forma, sin embargo, al usar una vista basada en clases podremos realizar ciertas modificaciones sobreescribiendo o llamando algunos atributos propios de éstas vistas, sin tener que escribir tanto código como en la vista basada en funciones.

¿Cuando usaremos vistas basadas en funciones y cuando usaremos vistas basadas en clases?

Esto será según la complejidad de las vistas del proyecto y la capacidad que tengamos de saber

DECLARANDO LA URL - URLS.PY

usar las vistas basadas en clases. Te recomendamos leer en la documentación de Django acerca de estas vistas (https://docs.djangoproject.com/en/4.2/topics/class-based-views/) para que puedas ver la cantidad de vistas pre-definidas que nos ofrece Django para facilitarnos la escritura de código. Pero repetimos, hay que saber que hace cada vista, para saber usarla.

Ahora usaremos esta vista basada en clases para "posts" por lo que también debemos definirlo en la url, por lo que vamos a dirigirnos a "urls.py" de nuestra aplicación y cambiamos el "path" de la vista basada en funciones por la forma de llamar a una vista basada en clases y que pueda renderizar el html.

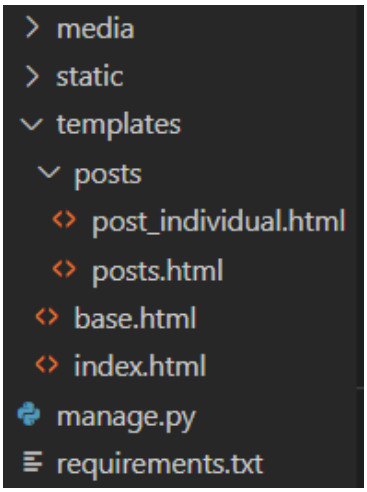
```
apps > posts > urls.py > ...
1   from django.urls import path
2   from .views import PostListView
3   from . import 
4
5
6   urlpatterns = [
7       path('posts/', PostListView.as_view(), name='posts'),
8   ]
```

Como puedes observar, importamos desde "views" la vista de clase "PostListView". En el "path" declaramos la ruta de acceso, luego la vista de clase (donde antes teníamos la vista en base a funciones) con la función ".as_view() y, por último, el nombre para hacer referencia a la url.
No olvides borrar la vista basada en funciones que teníamos anteriormente en "views.py", ya que esta no la usaremos más.
Guardamos todo y si lo probamos en el navegador se tiene que ver de la misma forma que se veía antes.
Ahora es momento de crear la vista para un registro individual, es decir, para acceder a un post en particular, por lo que vamos crear una view para el mismo, pero también debemos crear un nuevo html, el cual va a mostrar dicho post. Por lo que para comenzar a ordenar las plantillas que vamos a ir creando para lo que necesitemos mostrar. Entonces, vamos a crear una nueva carpeta dentro de la carpeta templates llamada "posts" para hacer referencia a estos. Dentro de esta carpeta vamos a

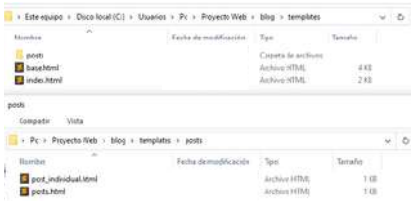
ACCEDIENDO MEDIANTE EL ID A UN REGISTRO EN PARTICULAR

PASOS

alojar el html para la vista del post individual y también vamos a mover el archivo "posts.html" que se encuentra en "templates" hacia la subcarpeta "posts" creada recientemente. La estructura quedaría así:



Como puedes observar, ahora los html "base" e "index" quedaron en la carpeta "templates" y "posts.html" y "post_individual.html" (que lo creamos) quedaron en la subcarpeta "posts". Así sería la vista desde el explorador de Windows:



Vamos a dirigirnos al views de nuestra aplicación para crear la vista basada en clases para nuestro post individual. Vas a ver también que debemos cambiar el acceso del template en la vista "PostListView" ya que cambiamos de carpeta a la plantilla "posts.html".

```
from .models import Post
from django.views.generic import ListView, DetailView

class PostListView(ListView):
    model = Post
    template_name = "posts/posts.html"
    context_object_name = "posts"

class PostDetailView(DetailView):
    model = Post
    template_name = "posts/post_individual.html"
    context_object_name = "posts"
    pk_url_kwarg = "id"
    queryset = Post.objects.all()
```

Para crear esta vista importamos la vista que nos provee Django "DetailView".
Creamos la vista de clase "PostDetailView" que hereda de "DetailView", donde al atributo "model" le decimos que busque en el modelo "Post", al atributo "template_name" que renderice la plantilla "posts.html", el atributo "pk_url_kwarg"

ACCEDIENDO MEDIANTE EL ID A UN REGISTRO EN PARTICULAR

PASOS

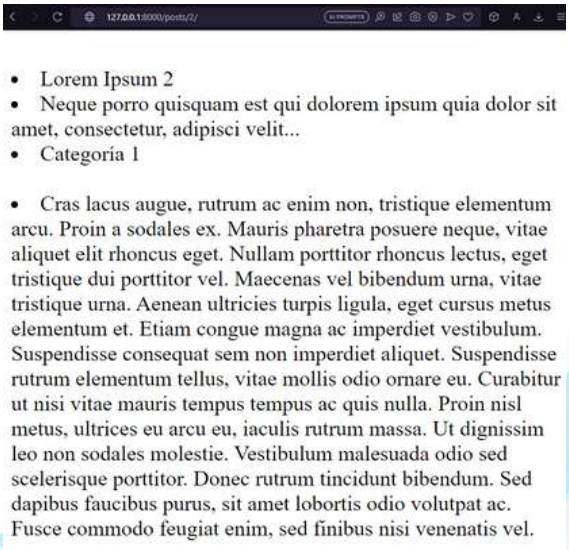
obtendrá el id del "post" y el atributo "queryset" se encarga de obtener todos los datos referidos al registro al que accedemos mediante el id obtenido.
Lo próximo es definir la url de acceso.

```
apps > posts > urls.py > ...
1   from django.urls import path
2   from .views import PostListView, PostDetailView
3
4   urlpatterns = [
5       path('posts/', PostListView.as_view(), name='posts'),
6       path("posts/<int:id>/", PostDetailView.as_view(), name="post_individual"),
7   ]
```

Para esta vista pasamos un argumento más que es el id obtenido y lo declaramos como un entero.
Ahora cargaremos nuestro template "post_individual.html" para poder mostrar un post individual.

```
templates > posts > post_individual.html > ...
1   <br>
2   <li>{{ posts.titulo }}</li>
3   <li>{{ posts.subtitulo }}</li>
4   <li>{{ posts.categoria }}</li>
5   <br>
6   <li>{{ posts.texto }}</li>
7
```

y la vista que tendríamos desde el navegador sería algo así (pusimos el id 2 para el ejemplo)



- Lorem Ipsum 2
- Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit...
- Categoría 1

- Cras lacus augue, rutrum ac enim non, tristique elementum arcu. Proin a sodales ex. Mauris pharetra posuere neque, vitae aliquet elit rhoncus eget. Nullam porttitor rhoncus lectus, eget tristique dui porttitor vel. Maecenas vel bibendum urna, vitae tristique urna. Aenean ultricies turpis ligula, eget cursus metus elementum et. Etiam congue magna ac imperdiet vestibulum. Suspendisse consequat sem non imperdiet aliquet. Suspendisse rutrum elementum tellus, vitae mollis odio ornare eu. Curabitur ut nisi vitae mauris tempus tempus ac quis nulla. Proin nisl metus, ultrices eu arcu eu, iaculis rutrum massa. Ut dignissim leo non sodales molestie. Vestibulum malesuada odio sed scelerisque porttitor. Donec rutrum tincidunt bibendum. Sed dapibus faucibus purus, sit amet lobortis odio volutpat ac. Fusce commodo feugiat enim, sed finibus nisi venenatis vel.

MOSTRANDO IMAGENES EN EL HTML

>>> PASOS

Para acceder a mostrar la imagen que tiene cargada nuestro post, tendremos que usar la siguiente declaración en template: **{{ posts.imagen.url }}**
Sin embargo, tenemos que realizar una modificación antes de poder usar esta llamada, ya que si la agregamos ahora, nos saldrá un ícono de imagen "rota" o "no encontrada".
La modificación que tenemos que realizar es la de declarar en nuestra "urls.py" principal, el siguiente código:

```
blog > urls.py > ...
17  from django.contrib import admin
18  from django.urls import path, include
19  from .views import index
20  from django.conf import settings
21  from django.conf.urls.static import static
22  from django.contrib.staticfiles.urls import staticfiles_urlpatterns
23
24
25  urlpatterns = [
26      path('admin/', admin.site.urls),
27      path('', index, name='index'),
28      path('', include('apps.posts.urls')),
29  ]+static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
30  urlpatterns += staticfiles_urlpatterns()
31  urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Lo primero es importar
from django.conf.urls.static import static
y luego
from django.contrib.staticfiles.urls import staticfiles_urlpatterns

las cuales van a servir para hacer las declaraciones:

+static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
urlpatterns += staticfiles_urlpatterns()
urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)

que debes ponerla tal cual está, con el + en la misma línea de código en la que termina la declaración de las urls (línea 29 en este caso.
Ahora te explicamos para que sirve esto:

- La primera (29 en este caso) línea usa la función "static" para añadir una URL que sirve los archivos estáticos desde la ruta definida en "settings.STATIC_URL" y que se encuentran en el directorio definido en "settings.STATIC_ROOT1". Esta función solo se debe usar en desarrollo, ya que en producción se espera que los archivos estáticos sean servidos por un servidor web.

MOSTRANDO IMAGENES EN EL HTML

PASOS

- La segunda línea (30 en este caso) usa la función "staticfiles_urlpatterns" para añadir las URLs que sirven los archivos estáticos desde las aplicaciones y los directorios definidos en "settings.STATICFILES_DIRS". Esta función es útil cuando se usa el sistema de archivos estáticos de Django, que permite recoger los archivos estáticos de cada aplicación y de otros lugares y colocarlos en un solo lugar que se pueda servir fácilmente.
- La tercera línea (31 en este caso) usa de nuevo la función static para añadir una URL que sirve los archivos multimedia desde la ruta definida en "settings.MEDIA_URL" y que se encuentran en el directorio definido en "settings.MEDIA_ROOT". Estos archivos son los que se suben por los usuarios, como imágenes, vídeos, documentos, etc. Al igual que con los archivos estáticos, esta función solo se debe usar en desarrollo.

Ahora podemos cargar en "post_individual.html" la etiqueta **{{ posts.imagen.url }}**

templates > posts > post_individual.html > img

```
<br>
<li>{{ posts.titulo }}</li>
<li>{{ posts.subtitulo }}</li>
<li>{{ posts.categoria }}</li>
<br>
<li>{{ posts.texto }}</li>
<img src="{{ posts.imagen.url }}" alt="">
```

Y en el navegador ya podremos ver la imagen del registro post al que estamos accediendo:

Ahora que todo está listo, ya que podemos acceder a los posts que están en nuestra base de datos y además podemos acceder a un post especifico, sería momento de enlazar todo haciendo las referencias desde el "base.html" para que al hacer click en posts, podamos ver el listado de posts y si luego hacemos click en un post en particular, podamos ver el detalle de este.

AGREGANDO LAS REFERENCIAS DE ACCESO A NUESTRAS URLS

PASOS

Para esto, lo primero que vamos a hacer es declarar el nombre de nuestra aplicación en "urls.py" de nuestra aplicación:

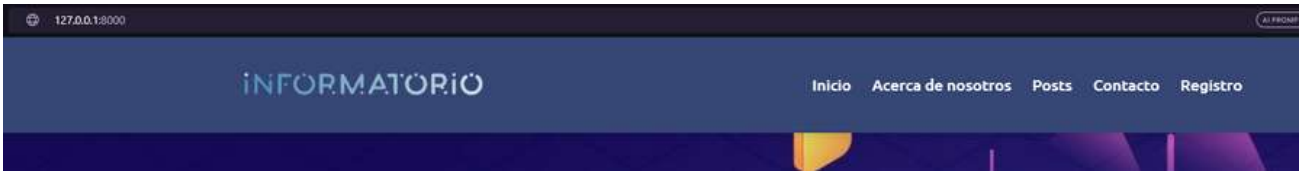
```
apps > posts > urls.py > ...
1   from django.urls import path
2   from . import views
3
4   app_name = 'apps.posts'
5
6   urlpatterns = [
7       path('posts/', views.PostListView.as_view(), name='posts'),
8       path("posts/<int:id>/", views.PostDetailView.as_view(), name="post_individual"),
9   ]
10
```

Hacemos esta declaración para que desde "base.hmtl" (o cualquier otro html) podamos hacer referencia mediante la sintaxis de Django a la aplicación en particular a la que queremos acceder desde un link o botón para acceder y además le decimos a la url especifica que vamos a acceder de la aplicación. Como puedes ver en este caso, tenemos dos urls declaradas y lo que queremos hacer es acceder a la url "posts" desde el Navbar, por lo que vamos a declarar en nuestro Navbar:

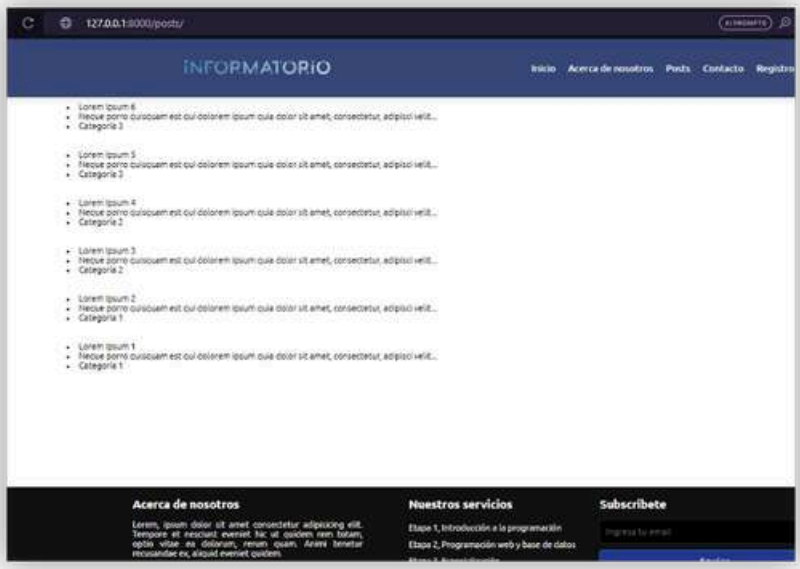
```
templates > base.html > html > body > header
33
34          <ul class="nav-links">
35            <li><a href="{% url 'index' %}">Inicio</a></li>
36            <li><a href="#">Acerca de nosotros</a></li>
37            <li><a href="{% url 'apps.posts:posts' %}">Posts</a></li>
38            <li><a href="#">Contacto</a></li>
39            <li><a href="#">Registro</a></li>
40          </ul>
41        </div>
42      </nav>
43
```

En esta estructura base de un Navbar delaramos en cada "botón" las referencias de acceso para cada plantilla en cuestión.
Hasta este momento del apunte tenemos creado el acceso de "index" desde el "urls.py" principal y ahora tenemos los accesos de la aplicación posts. A medida que vayamos creando más aplicaciones iremos haciendo más accesos y se irá completando nuestro Navbar y todos los accesos.
La forma en como se declara es básicamente con la nomenclatura que venimos usando. Para "index" será entonces **{% url 'index' %}**
y para nuestra app posts será:
{% url 'apps.posts:posts %}
donde apps se refiere a la carpeta donde está alojada nuestra aplicación, posts se refiere a la aplicación en sí, y :posts se refiere a la url creada para acceder a la vista que queremos, en este caso nos da el acceso a los posts alojados en la base de datos.
Ya podemos probrarlo:

AGREGANDO LAS REFERENCIAS DE ACCESO A NUESTRAS URLS



ingresamos a la página principal, damos click en Posts:



Ya accede y se ve correctamente los posts registrados.
Para que se aprecie mejor, hicimos herencia del template "base.html" tal cual ya lo habíamos hecho anteriormente.
Te vamos a dejar igualmente la captura de esto.
Pero lo que nos falta ahora para poder acceder a las noticias es darle una referencia y esto lo podemos hacer con un botón o simplemente asignar la referencia al título, o subtítulo. Vamos a hacerlo con un botón.

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

{% block contenido %}

<div class="container-fluid" style="margin: 200px;">

    {% for i in posts %}
        <br>
        <li>{{ i.titulo }}</li>
        <li>{{ i.subtitulo }}</li>
        <li>{{ i.categoria }}</li>
        <br>
        {% empty %}
        <h1>No hay registros</h1>
    {% endfor %}

</div>
{% endblock %}
```

A la izquierda la captura del template "post.html" y a la derecha agregamos el botón:

```
{% block contenido %}

<div class="container-fluid" style="margin: 200px;">

    {% for i in posts %}
        <br>
        <li>{{ i.titulo }}</li>
        <li>{{ i.subtitulo }}</li>
        <li>{{ i.categoria }}</li>
        <br>
        <a id="boton_post" href="{% url 'apps.posts:post_individual' i.id %}">
            Ingresá a este Post
        </a>

        {% empty %}
        <h1>No hay registros</h1>


    {% endfor %}
</div>
{% endblock %}
```

MUESTRA

MOSTRANDO EL RESULTADO

Puedes ver que para ingresar al post especifico lo referenciamos con
{% url 'apps.posts:post_individual' i.id %}
de manera tal que, a diferencia con la referencia al listado de posts, ahora ingresamos a través del **id** del post.
Agregamos algo de css ("boton_post") para que puedas ver mejor la forma de un botón simple.
Esta sería la vista final:

y la vista del ingreso al post un vez hecho click en el botón

Si seguiste los pasos hasta aquí (siempre respetando las sintaxis empleadas, los pasos correctos y guardando siempre cada cambio realizado) no deberías tener problemas y ver prácticamente como te estamos mostrando. Nosotros haremos los formatos con css y js para que se vaya viendo mejor.
¡Te invitamos a que también lo hagas! Luego te iremos mostrando que como verá nuestro blog.

Ahora comenzaremos con los **formularios**.

CREANDO EL FORMULARIO PARA CONTACTO

PASOS

Es el momento de crear nuestro primer formulario, el cual nos va a servir de formulario de contacto para nuestro blog.
Para el ejemplo, lo vamos a crear como una aplicación aparte, sin embargo, tu puedes hacerlo creando el modelo en una aplicación ya creada (en este momento la única aplicación creada es "posts", por lo que si quisieras podrías hacerlo ahí, sin embargo, para el ejemplo, te recomendamos seguir nuestros mismos pasos).
Vamos a posicionarnos en la carpeta apps y ejecutamos en consola: **django-admin startapp contacto**

```
(entorno) C:\Users\Pc\Proyecto Web\blog\apps>django-admin startapp contacto
```

Vamos a cargar el "models.py" de la aplicación creada.

```
apps > contacto > models.py > ...
1   from django.db import models
2   from django.utils import timezone
3
4   # Create your models here.
5
6   class Contacto(models.Model):
7       nombre_apellido = models.CharField(max_length=120)
8       email = models.EmailField()
9       asunto = models.CharField(max_length=50)
10      mensaje = models.TextField()
11      fecha = models.DateTimeField(default=timezone.now)
12
13      def __str__(self):
14          return self.nombre_apellido
15
```

Definimos campos en general usados como para

cualquier formulario de contacto.
Lo siguiente es cambiar el acceso en "apps.py" para decirle que busque en la carpeta "apps".

```
models.py   apps.py
apps > contacto > apps.py > ...
1   from django.apps import AppConfig
2
3
4   class ContactoConfig(AppConfig):
5       default_auto_field = 'django.db.models.BigAutoField'
6       name = 'apps.contacto'
7
```

Cambiamos

Lo registramos en "admin.py"

```
apps > contacto > admin.py > ...
1   from django.contrib import admin
2   from .models import Contacto
3
4   # Register your models here.
5
6
7   @admin.register(Contacto)
8   class ContactoAdmin(admin.ModelAdmin):
9       list_display = ('id', 'nombre_apellido', 'email', 'asunto', 'fecha')
10
```

El "list_display", lo configuramos para que aparezcan todos los campos definidos en nuestro modelo.
Vamos a "settings.py" y declaramos nuestra nueva aplicación.

CREANDO EL FORMULARIO PARA CONTACTO

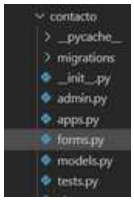
PASOS

```
ls.py    apps.py    admin.py    settings.py ×
onfiguraciones > settings.py > ...
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'apps.posts',
    'apps.contacto',
]
```

¡No olvides de ir guardando todos los cambios!
Con esto hecho, es momento de realizar por consola las migraciones.

```
(entorno) C:\Users\Pc\Proyecto Web\blog>python manage.py makemigrations
Migrations for 'contacto':
  apps\contacto\migrations\0001_initial.py
    - Create model Contacto

(entorno) C:\Users\Pc\Proyecto Web\blog>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contacto, contenttypes, posts, sessions
Running migrations:
  Applying contacto.0001_initial... OK
```

Es momento de crear nuestro formulario. Esto lo vamos a hacer dentro de la carpeta de nuestra aplicación como un nuevo archivo con el nombre de "**forms.py**".



Dentro de este archivo vamos a cargar todo los campos que declaramos en nuestro modelo pero con la herencia de un modelo de formulario propio de Django.

```
forms.py
blog > apps > contacto > forms.py > ...
from ... import ...
from .models import ...

class ...(forms.ModelForm):
    class Meta:
        model = ...
        fields = ['nombre_apellido', 'email', 'asunto', 'mensaje']
```

En las importaciones tenemos que traer "**forms**" desde Django y el modelo "Contacto".
Crearemos la clase, la cual nosotros pusimos como "ContactoForm", pero puede llevar cualquier nombre, sin embargo, te sugerimos que mantengas este tipo de referencias para que luego sea más fácil leer el código. Este formulario va a heredar de "ModelForm" a través de "form".
Luego definimos la clase "Meta" especificando el modelo y los campos.

CREANDO EL FORMULARIO PARA CONTACTO

PASOS

Ahora definiremos la view para este formulario:

```
forms.py   views.py   contacto.html
apps > contacto > views.py > ...
1   from .forms import ContactoForm
2   from django.contrib import messages
3   from django.views.generic import CreateView
4   from django.urls import reverse_lazy
5
6
7   class ContactoUsuario(CreateView):
8       template_name = 'contacto/contacto.html'
9       form_class = ContactoForm
10      success_url = reverse_lazy('index')
11
12      def form_valid(self, form):
13          messages.success(self.request, 'Consulta enviada.')
14          return super().form_valid(form)
15
```

Las importaciones que vamos a tener son las del form "ContactoForm", luego vamos a importar "messages" para mostrar un mensaje luego de enviar el contacto, también traemos otra vista que nos provee Django que es "CreateView" y por último usaremos "reverse_lazy" para que una vez enviado el contacto, nos devuelta a la página "index".
En este caso usamos "reverze_lazy" ya que utilizamos una vista de clases, sin embargo, puedes usar otros atributos url de vistas genéricas de Django. Te recomendamos leer la documentación de Django para que puedas comprender mejor estos atributos url (https://docs.djangoproject.com/en/4.2/ref/urlresolvers/)

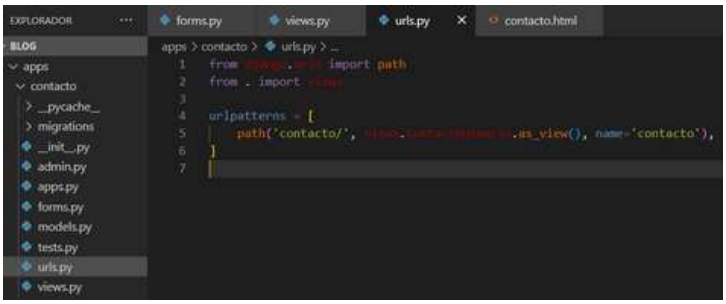
También debemos aclarar que esto es una vista de ejemplo muy básica. Si quieres hacer mejoras podrías usar el método "get_form_kwargs()" para pasar el request al formulario y así poder acceder al email del usuario autenticado si lo hay, también podrías sobrescribir el método "get_success_url()" para pasar el id del objeto creado al url de éxito y así poder mostrar los detalles del contacto o podrías usar el método "form_send_email()" para enviar el correo electrónico al destinatario y así separar la lógica de la vista del modelo.
Puede haber muchas mejoras pero te lo dejamos para que investigues (todo está en la documentación de Django) y poder hacer un mejor código con más funcionalidad.

Como puedes observar en la "view.py" la plantilla que vamos a usar la creamos con el nombre "contacto.html" dentro de una subcarpeta "contacto" que se encuentra en la carpeta "templates" - Estamos siguiendo la estructuración de carpetas que hicimos con los posts.

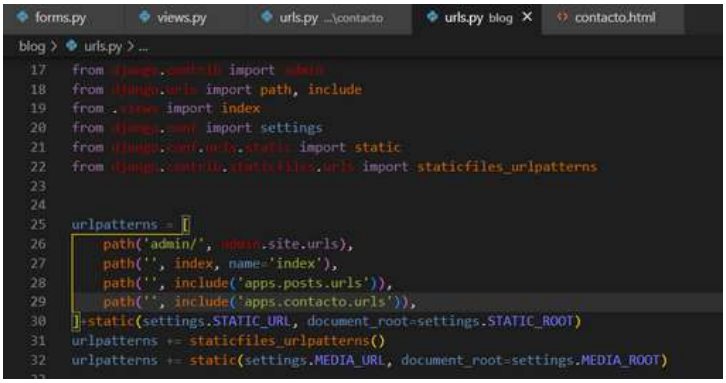
CREANDO EL FORMULARIO PARA CONTACTO

PASOS

Ahora crearemos la url, pero tenemos que crear el archivo "urls.py" dentro de nuestra aplicación, ya que si recuerdas, este archivo no se crea con la creación de la aplicación. Lo creamos y definimos el acceso:



Y antes de comenzar a completar el template "contacto.html" vamos a declarar esta url en el archivo "urls.py" principal.



Es momento de completar el template contacto.

```
templates > contacto > contacto.html > ...
1   {% extends 'base.html' %}
2   {% load static %}
3
4   {% block contenido %}
5
6   <center>
7       <form method="POST" style="margin: 200px;">
8           <h1>Formulario de contacto</h1>
9           <br>
10          <table>
11              {% csrf_token %}
12              {{ form }}
13          </table>
14          <input type="submit" value="enviar">
15      </form>
16  </center>
17
18  {% endblock %}
```

Esta es un forma muy simple de completarlo donde te damos la sintaxis básica de lo fundamental que debe ir el en html que son:

- form method="POST" ---> para poder usar POST que envía los datos en el cuerpo de la solicitud y no en la URL. El método POST es el más común para enviar formularios, especialmente cuando se trata de datos sensibles o que modifican el estado del servidor.

CREANDO EL FORMULARIO PARA CONTACTO

PASOS

- {% csrf_token %} es un tag de Django que protege el formulario contra ataques de falsificación de solicitudes entre sitios (CSRF, por sus siglas en inglés). Estos ataques ocurren cuando un sitio malicioso contiene un enlace, un botón o algún código javascript que intenta realizar alguna acción en tu sitio web, usando las credenciales de un usuario que visita el sitio malicioso en su navegador.
- {{ form }} trae los campos declarados en el formulario creado mediante la herencia del form que trajimos de Django.

Ahora si probamos, podemos enviar el formulario de contacto, queda recepcionado en el servidor y vuelve al "index", sin embargo, para usar el mensaje que definimos en la views mediante messages.success(self.request, 'Consulta enviada.')
tenemos que agregar algo más al html y definir un contexto en la view:

```
<form method="POST" style="margin: 200px;">
    <h1>Formulario de contacto</h1>
    <br>
    {% if messages %}
        {% for message in messages %}
            <div class="message {{ message.tags }}"> {{ message }} </div>
        {% endfor %}
    {% endif %}
    <table>
        {% csrf_token %}
        {{ form }}
    </table>
```

Lo que hacemos es introducir código Python para que

si hay mensajes, haga un recorrido en "messages" y lo muestre.
En views debemos agregar un contexto:

```
apps > contacto > views.py > ...
from django.urls import reverse_lazy


class ContactoUsuario(CreateView):
    template_name = 'contacto/contacto.html'
    form_class = ContactoForm
    success_url = reverse_lazy('apps.contacto:contacto')

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['request'] = self.request
        return context

    def form_valid(self, form):
        messages.success(self.request, 'Consulta enviada.')
        return super().form_valid(form)
```

Y como no estamos usando ni JS ni sweetalert o alguna otra forma de mostrar el mensaje de "Consulta enviada", en reverse_lazy le decimos que solo quede en "contacto" para que podamos visualizar el mensaje. Si recuerdas como lo hicimos en "posts", para acceder mediante esta declaración "apps.contacto:contacto", debemos declarar el nombre en "urls.py de nuestra aplicación "contacto":

MOSTRANDO RESULTADOS

PASOS

```
views.py | urls.py | contacto.html | forms.py
apps > contacto > urls.py > ...
1  from django.urls import path
2  from . import views
3
4  app_name = 'apps.contacto'
5
6  urlpatterns = [
7      path('contacto/', views.contactoUsuario.as_view(), name='contacto'),
8  ]
```

Volvemos a repetir, **no olvides guardar cada modificación hecha**, y ahora antes de probar y que quede todo accesible, en "base.html" vamos a hacer la referencia para poder acceder desde el Navbar a "Contacto":

```
views.py | urls.py | base.html | contacto.html | forms.py
templates > base.html > html > body > header > nav > div.nav-content > ul.nav-links
31          </div>
32          <ul class="nav-links">
33            <li><a href="{% url 'index' %}">Inicio</a></li>
34            <li><a href="#">Acerca de nosotros</a></li>
35            <li><a href="{% url 'apps.posts:posts' %}">Posts</a></li>
36            <li><a href="{% url 'apps.contacto:contacto' %}">Contacto</a></li>
37            <li><a href="#">Registro</a></li>
38          </ul>
39        </div>
40      </nav>
41
42
43      {% block contenido %}
44
45      {% endblock %}
46
```

Ahora desde nuestro "index" podemos acceder a Contacto. Enviamos una consulta y nos tiene que aparecer el mensaje de "Consulta enviada".
Y si accedemos al "admin" de Django, podremos ver el mensaje enviado.

Con esto realizamos todos los pasos para crear nuestro primer formulario. Algo parecido vamos a hacer para crear otros formularios como el de agregar posts para un usuario con acceso a esto, o para comentarios, también usaremos forms.
Hasta aquí llegamos con esta parte del apunte, en el próximo te mostraremos más de nuestra página formateada con css y js, por lo que si te animas, ¡también puedes hacerlo!

CREANDO APP USUARIO

PASOS

Para seguir integrando este blog, en este apartado veremos el uso de usuarios que nos provee Django. Para el contenido de este proyecto podemos usar User o AbstractUser o AbstractBaseUser.
Para nuestro caso vamos a usar AbstractUser, por lo que vamos a comenzar creando al app usuario para luego seguir con el modelo.

```
(entorno) C:\Users\Pc\Proyecto Web\blog\apps>django-admin startapp usuario
```

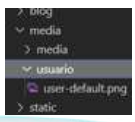
Ahora dentro de la aplicación usuario vamos a ir al "models.py"

```
apps > usuario > models.py > ...
from django.db import models
from django.urls import reverse
from django.contrib.auth.models import AbstractUser

# models

class Usuario(AbstractUser):
    imagen = models.ImageField(null=True, blank=True,upload_to='usuario', default='usuario/user-default.png')

    def get_absolute_url(self):
        return reverse('index')
```

Como puedes ver hacemos las importaciones pertinentes que usaremos. Dentro de estas está AbstractUser.
Al heredar de AbstractUser, el modelo Usuario incluye todos los campos y métodos del modelo de usuario predeterminado de Django, como nombre de usuario, correo electrónico y contraseña.
Además, en el modelo Usuario agregamos un campo adicional llamado "imagen". Este campo permite a los usuarios subir una imagen de perfil.
El campo imagen tiene las siguientes propiedades:

- null=True: Esto significa que el campo puede ser nulo en la base de datos.
- blank=True: Esto significa que el campo puede estar en blanco en los formularios.
- upload_to='usuario': Esto especifica el subdirectorio dentro del directorio MEDIA_ROOT donde se guardarán las imágenes subidas.
- default='usuario/user-default.png': Esto especifica la imagen predeterminada que se utilizará si el usuario no sube una imagen.
-

Finalmente, el modelo Usuario define un método llamado "get_absolute_url()". Este método devuelve la URL absoluta del objeto de usuario. En este caso, el método devuelve la URL correspondiente a la vista con el nombre 'index'. Esto significa que, cuando se llama al método get_absolute_url() en un objeto de usuario, se redirigirá al usuario a la página principal de la aplicación.
Para las imagenes de los usuarios, creamos una nueva carpeta "usuario" dentro de la carpeta "media" donde agregamos una imagen que va a ser la que lleve por defecto un usuario que se registre, si es que no sube una imagen propia, la cual, en tal caso, se guardará dentro de la carpeta "meida/usuario".



CONFIGURACIONES DE APP USUARIO

PASOS

Registraremos este modelo en el admin de Django, por lo que abrimos "admin.py" de nuestra app usuario:

```
admin.py
apps > usuario > admin.py
1   from django.contrib import admin
2   from .models import Usuario
3
4   # Register your models here.
5
6   admin.site.register(Usuario)
```

Luego en apps.py cambiaremos la ruta de acceso:

```
apps.py
apps > usuario > apps.py > ...
1   from django.apps import AppConfig
2
3   class UsuarioConfig(AppConfig):
4       default_auto_field = 'django.db.models.BigAutoField'
5       name = 'apps.usuario'
6
```

Lo próximo es configurar el settings.py agregando una nueva línea:

```
28  ALLOWED_HOSTS = []
29
30  AUTH_USER_MODEL    ='usuario.Usuario'
31
32  # Application definition
```

Luego agregamos "apps.usuario" en nuestras aplicaciones:

```
34  INSTALLED_APPS = [
35      'django.contrib.admin',
36      'django.contrib.auth',
37      'django.contrib.contenttypes',
38      'django.contrib.sessions',
39      'django.contrib.messages',
40      'django.contrib.staticfiles',
41
42      'apps.posts',
43      'apps.contacto',
44      'apps.usuario',
45  ]
```

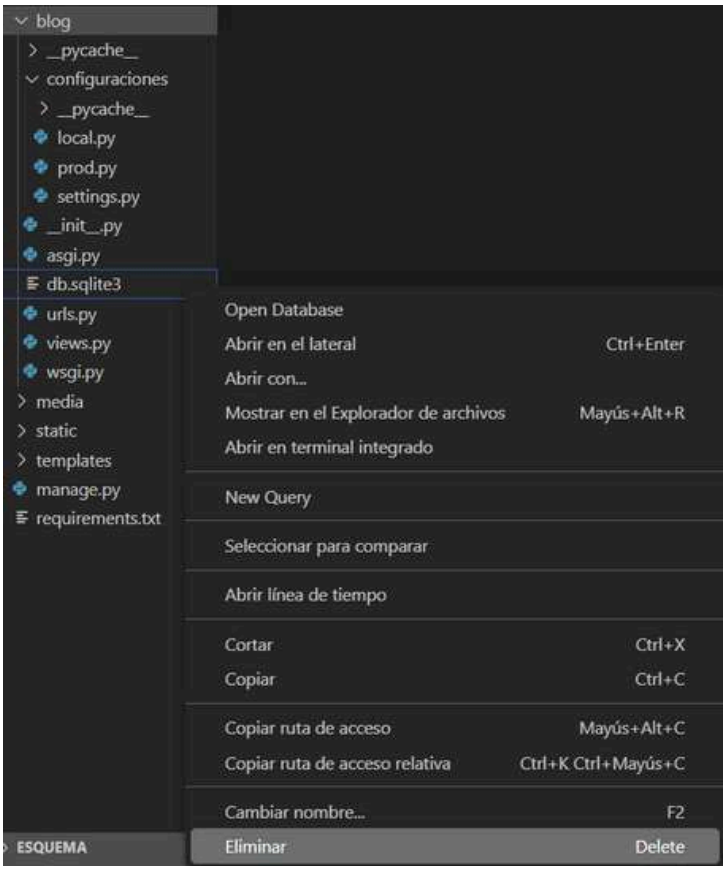
Ahora si quisieramos hacer las migraciones, nos arrojaría un error ya que hasta este momento estabamos utilizando el usuario "User" de Django, sin embargo, al crear un usuario "AbstractUser", estaríamos reemplazando el usuario actual de Django por lo que se generaría una inconsistencia en la base de datos. Por lo que es momento de enlazar con unos de los temas que los dejamos hasta este momento para que, con el nuevo tipo de usuario, puedas comenzar a usar la base de datos de MySql.

Pero antes de crear esta base de datos, y por si quieres seguir usando sqlite3, vamos a borrar los datos que ya tiene para poder usar AbstractUser.

BORRADO DE SQLITE3

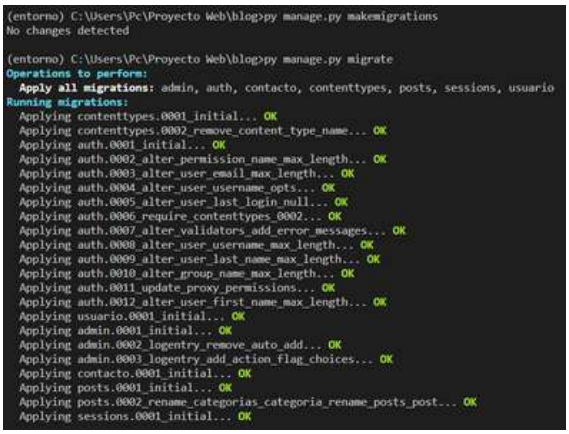
PASOS

Vamos a dirigirnos a la carpeta "blog/blog" y eliminamos nuestra base de datos manualmente con click derecho - Eliminar:



Nos saldrá una ventana donde nos dirá si estamos seguros que queremos eliminar y presionamos en "Mover a la papelera de reciclaje".

Una vez eliminada la base de datos de sqlite3, podemos realizar "makemigrations" y "migrate":



Ahora ya tenemos limpia nuestra base de datos y con "AbstractUser" funcionando en lugar de "User". Para poder volver a ingresar al admin de Django (http://127.0.0.1:8000/admin/) debemos crear nuevamente el superusuario como lo hicimos en la página 32. Hecho eso, ya podemos volver a acceder a nuestra base de datos sqlite3, sin embargo, como te mencionamos anteriormente, vamos a comenzar a usar MySQL y para esto te vamos a mostrar 2 formas de crear la base de datos.

MYSQL - CREACIÓN Y ENLACE CON EL PROYECTO

PASOS

Podemos hacerlo desde la consola o desde Workbench. Vamos a hacerlo desde la consola primero para que puedas ver las dos formas.
Primero vamos a abrir una consola cmd, ya sea desde Windows (Inicio - cmd) o desde la terminal en la que estamos trabajando desde VSC y ejecutamos mysql -u root -p

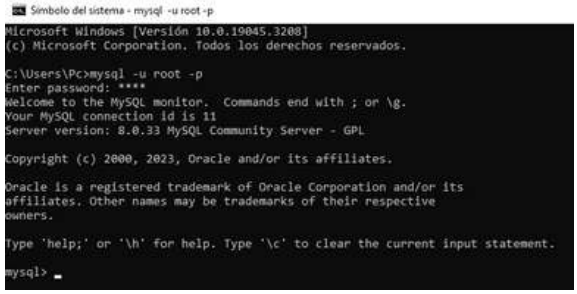
Si te llega a salir un error como este:

```
"mysql" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.
```

Debes realizar la siguiente configuración del sistema ya que esto se debe a que el "Path" del sistema no tiene agregado MySQL.

1. Abre el Panel de control y busca la opción “Sistema”.
2. Haz clic en “Configuración avanzada del sistema” en el menú de la izquierda.
3. Haz clic en el botón “Variables de entorno”.
4. En la sección “Variables del sistema”, busca la variable “Path” y haz clic en “Editar”.
5. Haz clic en “Nuevo” y agrega la ruta del directorio que contiene las herramientas de línea de comandos de MySQL (por ejemplo, C:\Program Files\MySQL\MySQL Server X.Y\bin).
6. Haz clic en “Aceptar” para guardar los cambios y cerrar todas las ventanas.

Si no hay ningún problema, la consola te pedirá la contraseña configurada.



Ahora ya podemos crear la base de datos como ya lo sabés hacer, con el comando CREATE seguido del nombre de la base de datos que vamos a crear. En este caso, vamos a llamarla con el mismo nombre del proyecto y agregando las letras db al final y luego para verla vamos a ejecutar el comando "show databases;" con el cual se mostrará la base de datos creada. Al final salimos con "exit".

```
Type 'help;' or '\h' for help. Type '

mysql> create database blogdb;
Query OK, 1 row affected (0.00 sec)

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| blogdb             |
```

MYSQL - CREACIÓN Y ENLACE CON EL PROYECTO

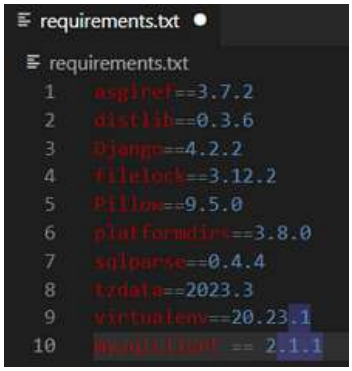
PASOS

Pero si queremos usar Workbench vamos a abrirlo y ejecutamos el mismo comando para crear nuestra base de datos.

Ahora es momento de modificar el settings.py de nuestro proyecto para comenzar a usar esta base de datos, pero, en nuestro caso, ahora comenzaremos a usar "local.py" de las configuraciones que habíamos hecho al principio. En este archivo declararemos que vamos a usar MySQL. Para ello necesitamos instalar la dependencia de MySQL para Django.
Vamos a la terminal y ejecutamos: **pip install mysqlclient**

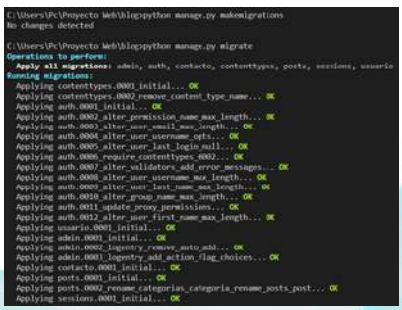
```
C:\Users\Pc\Proyecto Web\blog>pip install mysqlclient
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: mysqlclient in c:\users\pc\appdata\roaming\python\python311\site-packages (2.1.1)
```

y lo agregamos a requirements.txt. En este caso con la versión que se instaló, pero si usas otra versión, asegurate que sea la misma versión que se intaló al momento de hacer el proyecto.



Nos dirigimos a "local.py" y agregamos las siguientes líneas de código poniedo el password que usaremos (en nuestro caso usamos root).
Luego debes dirigirte a "settings.py" y borrar la base de datos declarada ahí:

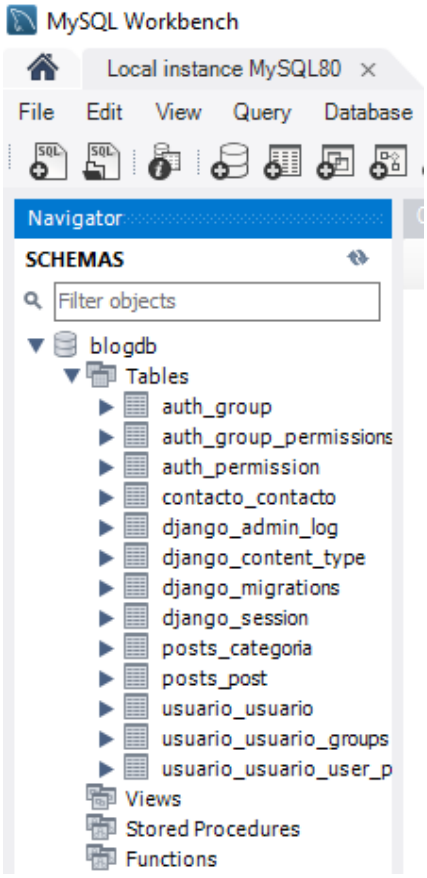
Guardamos todo y realizamos las migraciones:



MYSQL - CREACIÓN Y ENLACE CON EL PROYECTO

PASOS

Si miramos desde Workbench ahora (o desde la consola cmd), vamos a ver que se crearon correctamente las tablas:



Ahora nuevamente debemos crear el superuser para esta base de datos (página 32) para poder acceder al admin de Django.

No olvides que teníamos el servidor detenido para hacer estas modificaciones y tienes que volver a correrlo para ingresar al admin nuevamente.

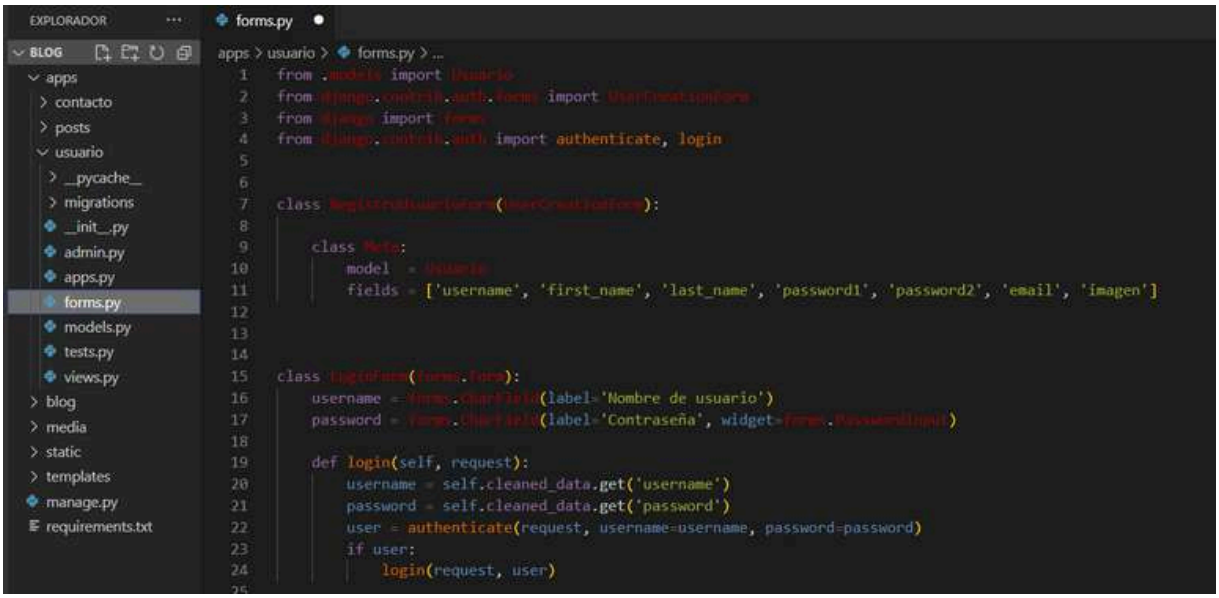
Como verás ahora se separó Usuarios de Grupos y esto es porque ahora estamos usando el AbstractUser para declarar un nuevo usuario en las apps, en vez de User que estuvimos usando hasta el momento.

Ahora podemos seguir personalizando nuestro usuario.

FORMS - VIEW - URL - TEMPLATE PARA USUARIOS

PASOS

Es momento de crear un login de usuario por lo que necesitamos un formulario para esto y, así que como hicimos con "contacto" vamos a crear un forms.py en nuestra app usuario:



Como puedes ver este código es un ejemplo de cómo se pueden crear formularios en Django para el registro y el inicio de sesión de usuarios.

Primero, se importan las clases necesarias de Django y del modelo Usuario. Luego, se define la clase "RegistroUsuarioForm" que hereda de "UserCreationForm" y se especifica que el modelo a utilizar es "Usuario" y los campos que se van a incluir en el formulario. Después, se define la clase "LoginForm" que hereda de forms.Form y se definen los campos para el nombre de usuario y la contraseña. Finalmente, se define el método "login" que autentica al usuario e inicia sesión si las credenciales son correctas.

El siguiente paso, como lo venimos haciendo, es crear las vistas. En estas vistas basadas en clases que vamos a usar, vamos a estar heredando desde lo que nos provee Django (como habrás leído sobre estas vistas cuando te lo recomendamos en la página 51).

>>> USUARIO

FORMS - VIEW - URL - TEMPLATE PARA USUARIOS

>>> PASOS

```
apps > usuario > views.py > ...
1   from .forms import RegistroUsuarioForm
2   from django.contrib.auth.views import LoginView, LogoutView
3   from django.views.generic import CreateView
4   from django.contrib import messages
5   from django.shortcuts import redirect
6   from django.urls import reverse
7
8   # Create your views here.
9
10  class RegistrarUsuario(CreateView):
11      template_name = 'registration/registrar.html'
12      form_class = RegistroUsuarioForm
13
14      def form_valid(self, form):
15          messages.success(self.request, 'Registro exitoso. Por favor, inicia sesión.')
16          form.save()
17
18          return redirect('apps.usuario:registrar')
19
20  class LoginUsuario(LoginView):
21      template_name = 'registration/login.html'
22
23      def get_success_url(self):
24          messages.success(self.request, 'Login exitoso')
25
26          return reverse('apps.usuario:login')
27
28
29  class LogoutUsuario(LogoutView):
30      template_name = 'registration/logout.html'
31
32      def get_success_url(self):
33          messages.success(self.request, 'Logout exitoso')
34
35          return reverse('apps.usuario:logout')
36
```

Para registrar el usuario heredamos de la vista CreateView, como lo hicimos con "contacto" y, de la misma forma también usamos el método para guardar si el formulario es válido con un mensaje incluído. La redirección de la página la dejamos en el registro para poder ver el mensaje de éxito, sin embargo, cuando mejores el código en tu proyecto puedes usar, por ejemplo, SweetAlert para mostrar un mensaje emergente del registro exitoso y que se redirija a la página de iniciar sesión (puedes buscar en la documentación de Django e investigar sobre mensajes: https://docs.djangoproject.com/en/4.2/ref/contrib/messages/ y sobre mensajes SweeAlert: https://lipis.github.io/bootstrap-sweetalert/). Lo mismo puedes hacer para "login", "logout" o "contacto" e incluso para "comentarios" (app que crearemos luego).

FORMS - VIEW - URL - TEMPLATE PARA USUARIOS

PASOS

Luego para "login" heredamos de la vista que nos proporciona Django "LoginView" donde solo agregamos un mensaje de "Login exitoso" y lo mismo hacemos para "logout". En ambos casos volvemos a quedar en el mismo template para ver los mensajes, pero la idea es que se redirija a la página de inicio, o si estamos en una cierta página, como sería la de un post en particular y queremos hacer un comentario pero debemos loguearnos o registrarnos, debería volver a la misma página del post en el que estamos. Puedes investigar sobre "request.path" (?next={{ request.path }} - https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication).

Esto último se agrega sobre los templates, por lo que en estos enlaces que te dejamos, puedes investigar, sin embargo, ahora nos toca realizar los templates (de forma muy básica) para poder visualizar lo que estuvimos haciendo.

Como pudiste ver en las views.py vamos a usar los templates registrar.html, login.html y logout.html que se encuentran en la carpeta "registration" la cual es una convención de Django ya que estamos usando las vistas que nos provee:

Dentro de estas tres plantillas vamos a usar formularios como lo habíamos hecho con "contacto", ya que, para todos necesitamos trabajar con la base de datos, por lo que llevarán las mismas bases.

Además, como lo hicimos en "contacto" agregaremos código Python para recorrer los mensajes y nos devuelva los mismos.

Te dejamos las vistas http para que veas la forma básica y luego los templates de cada uno.

FORMS - VIEW - URL - TEMPLATE PARA USUARIOS

Por el momento venimos escribiendo el nombre de la url de forma manual en el http (http://127.0.0.1:8000/logout/), pero luego haremos los enlaces y te lo mostraremos. Pero antes te dejamos las plantillas html:

templates > registration > registrar.html > ...

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

<title>{% block title %} Registro de usuario {% endblock title %}</title>

{% block contenido %}
<center>

    <form method="POST" enctype="multipart/form-data" style="margin-top: 300px; margin-bottom: 100px;">
        <h1 class="h1">Registro de usuarios</h1>
        <br>
        <div class="messages">
            {% for message in messages %}
                <table{% if message.tags %} class="{{ message.tags }}"{% endif %}>{{ message }}</table>
            {% endfor %}
        </div>
        <br>
        {% csrf_token %}
        <div class="container-fluid">
            <table>{{ form }}</table>
        </div>
        <br><br>
        <button type="submit" class="btn btn-success btn-lg px-3">Registrar</button>
    </form>

</center>
{% endblock %}
```

Para registrar es muy importante que agregues **enctype="multipart/form-data"** ya que sin este no subirá la imagen cuando la agreguemos en el campo "imagen". El resto del código, como verás es muy similar al de "contacto", por lo que cada vez que queramos un formulario, usaremos básicamente esta estructura. Hay veces que necesitaremos campos específicos y en esos casos utilizaremos "label_tag" (https://docs.djangoproject.com/en/4.2/ref/forms/api/).

templates > registration > logout.html > ...

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

{% block contenido %}
<center>
<div class="container" style="margin-top: 370px;">
    <div class="messages">
        {% for message in messages %}
            <table{% if message.tags %} class="{{ message.tags }}"{% endif %}>{{ message }}</table>
        {% endfor %}
    </div>
</div>
</center>
{% endblock %}
```

Logout no llevará demasiado código ya que lo más importante aquí es mostrar el mensaje de logout. Django se ha encargado de realizar el logout correctamente.

FORMS - VIEW - URL - TEMPLATE PARA USUARIOS

```
templates > registration > login.html > ...
{% extends 'base.html' %}
{% load static %}

<title>{% block title %} Registro de usuario {% endblock title %}</title>

{% block contenido %}
<center>

    <form method="POST" style="margin-top: 300px;">
        <h1 class="h1">Iniciar sesión</h1>
        <br>
        <div class="messages">
            {% for message in messages %}
                <table{% if message.tags %} class="{{ message.tags }}"{% endif %}>{{ message }}</table>
            {% endfor %}
        </div>
        <br>
        {% csrf_token %}
        <div class="container">
            <table>{{ form }}</table>
            <br>
            <a style="color: black;" href="{% url 'apps.usuario:registrar' %}">Todavía no está registrado? Registrate</a>
            <br><br>
            <button type="submit" class="btn btn-success btn-lg px-3">Iniciar sesión</button>
        </div>
    </form>

</center>
{% endblock %}
```

Y para login lo que hicimos fue agregar un botón para que si el usuario no está registrado, pueda acceder directamente al registro de usuario.

Ahora solo nos resta configurar en base.html los accesos desde el Navbar:

```
templates > base.html > html > body > header > nav > div.nav-content > ul.nav-links > li
        </div>
        <ul class="nav-links">
          <li><a href="{% url 'index' %}">Inicio</a></li>
          <li><a href="#">Acerca de nosotros</a></li>
          <li><a href="{% url 'apps.posts:posts' %}">Posts</a></li>
          <li><a href="{% url 'apps.contacto:contacto' %}">Contacto</a></li>
          <li><a href="{% url 'apps.usuario:login' %}">Login</a></li>
        </ul>
      </div>
```

Con esto podremos acceder a loguearnos con nuestro usuario, pero si el usuario no se encuentra registrado, puede acceder desde el mismo template de login al registro de usuarios.

Sin embargo, nos falta una cláusula más para que se habilite un botón de "logout" cuando el usuario está logueado, y eso lo vamos a hacer introduciendo un condicional a nuestro html: (lo vemos en la siguiente página)

FORMS - VIEW - URL - TEMPLATE PARA USUARIOS

templates > base.html > html > body > header > nav > div.nav-content > ul.nav-links

```
31          </div>
32          <ul class="nav-links">
33            <li><a href="{% url 'index' %}">Inicio</a></li>
34            <li><a href="#">Acerca de nosotros</a></li>
35            <li><a href="{% url 'apps.posts:posts' %}">Posts</a></li>
36            <li><a href="{% url 'apps.contacto:contacto' %}">Contacto</a></li>
37            {% if user.is_authenticated %}
38              <li><a href="{% url 'apps.usuario:logout' %}">Logout</a></li>
39            {% else %}
40              <li><a href="{% url 'apps.usuario:login' %}">Login</a></li>
41            {% endif %}
42          </ul>
43        </div>
44      </nav>
```

Lo que estamos haciendo en este momento, es decirle a Django que verifique que si el usuario está loguedo o autenticado, el botón que tiene que aparecer es el de "Logout", pero si no está logueado, tiene que aparecer "Login".

Lo siguiente para finalizar con los usuarios es hacer los accesos de reseto de password por si el usuario se ha olvidado la contraseña.
Esto lo vamos a hacer gracias a que Django por defecto ya tiene instalada la aplicación:

```
33
34  INSTALLED_APPS = [
35      'django.contrib.admin',
36      'django.contrib.auth',
37      'django.contrib.contenttypes',
38      'django.contrib.sessions',
39      'django.contrib.messages',
40      'django.contrib.staticfiles',
41
42      'apps.posts',
43      'apps.contacto'
```

Mediante algunas pequeñas configuraciones, vamos a tener el reseteo completo del password del usuario.

Lo primero va a ser agregar en la urls.py principal lo siguiente:

blog > urls.py > ...

```
18  from django.urls import path, include
19  from .views import index
20  from django.conf import settings
21  from django.conf.urls.static import static
22  from django.contrib.staticfiles.urls import staticfiles_urlpatterns
23
24
25  urlpatterns = [
26      path('admin/', admin.site.urls),
27      path('', index, name='index'),
28      path('', include('apps.posts.urls')),
29      path('', include('apps.contacto.urls')),
30      path('', include('apps.usuario.urls')),
31      path('', include('django.contrib.auth.urls'))
32  ]+static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
33  urlpatterns += staticfiles_urlpatterns()
34  urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
35
```

```
31      path('', include('django.contrib.auth.urls'))
```

FORMS - VIEW - URL - TEMPLATE PARA USUARIOS

Lo siguiente será crear las urls desde la aplicación usuario:

```
apps > usuario > urls.py > ...
1   from django.urls import path
2   from . import views
3   from .views import loginUsuario
4   from django.contrib.auth import views as auth_views
5
6   app_name = 'apps.usuario'
7
8   urlpatterns = [
9       path('registrar/', views.RegistrarUsuario.as_view(), name='registrar'),
10      path('login/', loginUsuario.as_view(), name='login'),
11      path('logout/', views.logoutUsuario.as_view(), name='logout'),
12      path('password_reset/', auth_views.PasswordResetView.as_view(), name='password_reset'),
13      path('password_reset/done/', auth_views.PasswordResetDoneView.as_view(), name='password_reset_done'),
14      path('reset/<uidb64>/<token>/', auth_views.PasswordResetConfirmView.as_view(), name='password_reset_confirm'),
15      path('reset/done/', auth_views.PasswordResetCompleteView.as_view(), name='password_reset_complete'),
16  ]
```

donde tendremos que agregar los "path" para "password_reset", "password_reset/done", "reset/<uidb64>/<token>/" y "reset/done/". Posterior a esto debemos crear las plantillas para mostrar los formularios y mensajes correspondientes:

- registration/password_reset_form.html: Esta plantilla se utiliza para mostrar el formulario donde los usuarios pueden ingresar su dirección de correo electrónico para solicitar un restablecimiento de contraseña.
- registration/password_reset_done.html: Esta plantilla se utiliza para mostrar un mensaje indicando que se ha enviado un correo electrónico con instrucciones para restablecer la contraseña.
- registration/password_reset_email.html: Esta plantilla se utiliza para generar el correo electrónico que se envía al usuario con instrucciones para restablecer su contraseña. Debe incluir un enlace a la URL password_reset_confirm con los argumentos uidb64 y token proporcionados por la vista.
- registration/password_reset_confirm.html: Esta plantilla se utiliza para mostrar el formulario donde los usuarios pueden ingresar una nueva contraseña.
- registration/password_reset_complete.html: Esta plantilla se utiliza para mostrar un mensaje indicando que la contraseña se ha restablecido correctamente.

Por último, antes de crear y completar cada template, debemos configurar un correo electrónico

FORMS - VIEW - URL - TEMPLATE PARA USUARIOS

para que se realice el envío del formulario de reseteo de la contraseña. Esta configuración requiere un correo que usemos para hacer el envío del formulario. Si vamos a utilizar un correo distinto cuando trabajemos en producción, te recomendamos realizar la configuración en local.py de nuestras configuraciones y en prod.py el correo que utilizaremos cuando este proyecto se encuentre en producción.
Si vas a usar el mismo correo tanto para el entorno loca, como para el entorno de producción, puedes poner esta cofiguración directamente en settings.py. Nosotros vamos a agregarla en settings.py para el ejemplo.
Puedes agregar estas líneas de código en cualquier parte de tu archivo de configuración:

blog > configuraciones > settings.py > ...

```
30  AUTH_USER_MODEL    ='usuario.Usuario'
31
32  EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
33  EMAIL_HOST = 'smtp.gmail.com'
34  EMAIL_PORT = 587
35  EMAIL_USE_TLS = True
36  EMAIL_HOST_USER = 'email_a_utilizar@gmail.com'
37  EMAIL_HOST_PASSWORD = 'contraseña de tu email'
38
39  # Application definition
40
41  INSTALLED_APPS = [
```

Asegurate de reemplazar por tu email y contraseña en esta parte.

Es momento de crear y completar nuestras plantillas para mostrar los formularios y mensajes. Lo vemos en la siguiente página.

FORMS - VIEW - URL - TEMPLATE PARA USUARIOS

password_reset_form.html

templates > registration > password_reset_form.html > ...

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

{% block contenido %}
  <h2>Restablecer contraseña</h2>
  <form method="POST">
    {% csrf_token %}
    {{ form }}
    <button type="submit">Enviar correo electrónico</button>
  </form>
{% endblock %}
<center>
    <div class="container" style="margin-top: 370px;">
        <div class="messages">
            {% for message in messages %}
                <table{% if message.tags %} class="{{ message.tags }}"{% endif %}>{{ message }}</table>
            {% endfor %}
        </div>
    </div>
</center>
```

password_reset_done.html

templates > registration > password_reset_done.html > ...

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

{% block contenido %}
<center>
  <h2>Correo electrónico enviado</h2>
  <p>Se ha enviado un correo electrónico con instrucciones para restablecer tu contraseña.</p>
</center>
{% endblock %}
```

FORMS - VIEW - URL - TEMPLATE PARA USUARIOS

password_reset_email.html

templates > registration > password_reset_email.html

```
Hola, has solicitado restablecer tu contraseña. Haz clic en el siguiente enlace para completar el proceso:

{{ protocol }}://{{ domain }}{% url 'password_reset_confirm' uidb64=uid token=token %}

Si no has solicitado restablecer tu contraseña, ignora este correo electrónico.

Saludos,
El equipo de {{ site_name }}
```

Aquí no hay que cambiar nada. Django se encargará de todo.

- {{ protocol }}: Esta variable se reemplaza por el protocolo utilizado para generar el enlace de restablecimiento de contraseña (por ejemplo, “http” o “https”).
- {{ domain }}: Esta variable se reemplaza por el nombre de dominio de tu sitio (por ejemplo, “www.example.com”).
- {% url 'password_reset_confirm' uidb64=uid token=token %}: Esta etiqueta de plantilla genera la URL para la vista de confirmación de restablecimiento de contraseña. Las variables uidb64 y token se reemplazan automáticamente por los valores correctos para el usuario que solicitó el restablecimiento de contraseña.

En cuanto a la variable {{ site_name }}, esta variable se reemplaza por el nombre de tu sitio. Puedes definir el valor de esta variable en la configuración SITE_NAME de tu proyecto de Django. Por ejemplo, si deseas que el nombre de tu sitio sea “Mi sitio”, puedes agregar la siguiente línea a tu archivo settings.py:

settings.py

blog > configuraciones > settings.py > ...

```
38
39    SITE_NAME = 'Informatorio'
40
```

FORMS - VIEW - URL - TEMPLATE PARA USUARIOS

password_reset_confirm.html

templates > registration > password_reset_confirm.html > ...

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

{% block contenido %}
<center>
  <h2>Restablecer contraseña</h2>
  <form method="POST">
    {% csrf_token %}
    {{ form }}
    <button type="submit">Restablecer contraseña</button>
  </form>
</center>
{% endblock %}
```

password_reset_complete.html

templates > registration > password_reset_complete.html > ...

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

{% block contenido %}
<center>
  <h2>Contraseña restablecida</h2>
  <p>Tu contraseña se ha restablecido correctamente.</p>
</center>
{% endblock %}
```

Hecho esto, solo queda enlazar las urls correspondiente en el template de login, arriba o abajo de registrarse (lo que comunmente se hace) y con eso ya se completaria la parte básica de usuario.
¡Te lo dejamos para que lo hagas!

COMENTARIOS SOBRE LOS POST

>>> PASOS

Es momento de crear los comentarios para los post. Esto se lo puede hacer desde una aplicación nueva o desde una aplicacion existente. Para este caso y simplificarlo más, nosotros vamos a usar la aplicación "post" para agregar los comentarios ahí mismo, por lo que en primera instancia vamos a cargar el modelo en models.py de la app "post". Importamos "seetings" que vamos a usar para autenticar a los usuarios:

```
apps > posts > models.py > ...
1   from django.db import models
2   from django.utils import timezone
3   from django.conf import settings
4
```

Creamos el modelo:

```
apps > posts > models.py > ...
35
36  class Comentario(models.Model):
37      posts = models.ForeignKey('posts.Post', on_delete=models.CASCADE, related_name='comentarios')
38      usuario = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE, related_name='comentarios')
39      texto = models.TextField()
40      fecha = models.DateTimeField(auto_now_add=True)
41
42      def __str__(self):
43          return self.texto
```

Donde "posts" es el campo de clave foránea que relaciona cada comentario con un post y si se elimina un post, todos los comentarios relacionados con ese post también se eliminarán debido a la opción on_delete=models.CASCADE.
"usuario" es otro campo de clave foránea que relaciona cada comentario con un usuario. Utiliza el modelo de usuario especificado en la configuración AUTH_USER_MODEL y establece una relación de “muchos a uno” entre comentarios y usuarios. Cuando se elimina un usuario, todos los comentarios relacionados con ese usuario también se eliminarán debido a la opción on_delete=models.CASCADE.
El resto de campo, como el método __str__ lo basamos en modelos anteriores que creamos.
Lo siguiente que haremos será registrar "Comentario" (previa importación desde el modelo " from .models import Categoria, Post, Comentario ") en el administrador de Django desde admin.py:

```
apps > posts > admin.py > ...
16  admin.site.register(Comentario)
```

COMENTARIOS SOBRE LOS POST

PASOS

Ahora crearemos un nuevo archivo "forms.py" como lo venimos haciendo en las aplicaciones de contacto y usuario, pero ahora lo haremos en la aplicación posts, y lo cargamos:

apps > posts > forms.py > ...

```python
from django import forms
from .models import Comentario

class ComentarioForm(forms.ModelForm):
    class Meta:
        model = Comentario
        fields = ['texto']
```

Para la vista actualizaremos la vista de "PostDetailView" agregando una función y luego crearemos una nueva vista para comentario (hacemos la importación del form que creamos " from .models import Post, Comentario " y la importación de redirect " from django.shortcuts import redirect "):

apps > posts > views.py >

```python
class PostDetailView(DetailView):
    model = Post
    template_name = "posts/post_individual.html"
    context_object_name = "posts"
    pk_url_kwarg = "id"
    queryset = Post.objects.all()

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['form'] = ComentarioForm()
        context['comentarios'] = Comentario.objects.filter(posts_id=self.kwargs['id'])
        return context

    def post(self, request, *args, **kwargs):
        form = ComentarioForm(request.POST)
        if form.is_valid():
            comentario = form.save(commit=False)
            comentario.usuario = request.user
            comentario.posts_id = self.kwargs['id']
            comentario.save()
            return redirect('apps.posts:post_individual', id=self.kwargs['id'])
        else:
            context = self.get_context_data(**kwargs)
            context['form'] = form
            return self.render_to_response(context)
```

>>> COMENTARIO

COMENTARIOS SOBRE LOS POST

>>> PASOS

Esta función es un método post que se agregamos a la vista que ya teníamos creada para manejar solicitudes POST. Cuando el usuario envíe el formulario de comentario con solicitud POST al servidor con los datos del formulario, este método procesará esa solicitud y realizará las acciones necesarias para guardar los datos del formulario en la base de datos.

```
def post(self, request, *args, **kwargs):
    >> Crea una instancia del formulario ComentarioForm con los datos enviados en la solicitud POST
    form = ComentarioForm(request.POST)

     >> Verifica si el formulario es válido (es decir, si todos los campos requeridos están presentes y son válidos)
     if form.is_valid():
        >> Si el formulario es válido, guarda los datos del formulario en la base de datos
        comentario = form.save(commit=False)
        comentario.usuario = request.user
        comentario.posts_id = self.kwargs['id']
        comentario.save()

     >> Redirige al usuario a la vista de detalle del post
        return redirect('apps.posts:post_individual', id=self.kwargs['id'])
    else:
        >> Si el formulario no es válido, vuelve a mostrar el formulario con los errores de validación
        context = self.get_context_data(**kwargs)
        context['form'] = form
        return self.render_to_response(context)
```

Además en el método "get_context_data" agregamos un campo más de contexto para manejar los comentarios que ya están creados y poder mostrarlos:

```
context['comentarios'] = Comentario.objects.filter(posts_id=self.kwargs['id'])
```

Luego creamos la vista de Comentario:

COMENTARIO

COMENTARIOS SOBRE LOS POST

PASOS

```
apps > posts > views.py > ...
39  class ComentarioCreateView(LoginRequiredMixin, CreateView):
40      model = Comentario
41      form_class = ComentarioForm
42      template_name = 'comentario/agregarComentario.html'
43      success_url = 'comentario/comentarios/'
44
45      def form_valid(self, form):
46          form.instance.usuario = self.request.user
47          form.instance.posts_id = self.kwargs['posts_id']
48          return super().form_valid(form)
```

"ComentarioCreateView" hereda de "LoginRequiredMixin" y "CreateView". "LoginRequiredMixin" es un mixin que requiere que el usuario esté autenticado para acceder a la vista. Si el usuario no está autenticado, será redirigido a la página de inicio de sesión. "CreateView" la hemos visto antes la cual es una vista genérica de Django que proporciona una forma fácil de crear objetos en la base de datos utilizando un formulario (las importaciones son: " from django.contrib.auth.mixins import LoginRequiredMixin " y " from django.views.generic.edit import CreateView ").

La vista "ComentarioCreateView" está configurada para utilizar el modelo "Comentario" y el formulario "ComentarioForm". Cuando un usuario accede a esta vista, se mostrará un formulario para crear un nuevo comentario utilizando el formulario "ComentarioForm". Cuando el usuario envía el formulario, los datos del formulario se utilizarán para crear un nuevo objeto "Comentario" en la base de datos.

El método form_valid se utiliza para establecer el usuario y el post asociados con el comentario antes de guardarlo en la base de datos. El usuario se establece en el usuario autenticado actual (self.request.user) y el post se establece en el valor del parámetro "posts_id" pasado en la URL.

Una vez que se ha creado el comentario, el usuario es redirigido a la página actual con el comentario creado. Esto lo vamos a incluir en el template de "post_individual.html"

```
templates > posts > post_individual.html > ...
1   {% extends 'base.html' %}
2   {% load static %}
3
4   {% block contenido %}
5   <center>
6   <div class="container-fluid" style="margin: 200px;">
7
8       <table>{{ posts.titulo }}</table>
9       <table>{{ posts.subtitulo }}</table>
10      <table>{{ posts.categoria }}</table>
11      <br>
12      <table>{{ posts.texto }}</table>
13      <img src="{{ posts.imagen.url }}" alt="">
14
15  </div>
16  </center>
```

COMENTARIOS SOBRE LOS POST

PASOS

templates > posts > post_individual.html > ...

```
<center>
<div class="container-fluid" style="margin-bottom: 20px;">
    <h4>Comentarios</h4>
    <br><br>
</div>
<div class="container-fluid" style="margin-bottom: 20px;"></div>
    {% for comentario in comentarios %}
        <table>{{ comentario.usuario }} - {{ comentario.fecha }}</table>
        <ul class="w-100 list-unstyled d-flex justify-content-between mb-0">
            <p>{{ comentario.texto }}</p>
            <br><br>
        </ul>
    {% empty %}
    <li>No hay comentarios - Puedes ser el primero en comentar!</li>
    {% endfor %}
</div>
<a id="comentario"></a>
<div class="container-fluid" style="margin-bottom: 100px;">
<form method="POST" style="margin-bottom: 100px; margin-top: 100px;">
    {% csrf_token %}
    {% if user.is_authenticated %}
    <h2>Deja tu comentario</h2>
        <form method="POST">
            {% csrf_token %}
            {{ form.as_p }}
            <input type="submit" value="Comentar">
        </form>
    {% else %}
        <h2>Debes iniciar sesión o registrarte para comentar</h2>
        <a class="btn btn-success btn-lg"  href="{% url 'apps.usuario:login' %}?next={{ request.path }}#comentario">Iniciar sesión</a>
        <input type="hidden" name="next" value="{{ request.path }}">
        {% endif %}
</form>
</div>
</center>
{% endblock %}
```

Por último, antes de poder ejecutar el servidor debemos hacer las migraciones (python manage.py makemigrations y python manage.py migrate) ya que realizamos cambios en el modelo de "post".

Ahora ya tienes las bases para la creación de comentarios. Puedes adaptarlo a tu template para que vaya quedando visiblemente mejor.

COMENTARIO

MOSTRANDO RESULTADOS

PASOS

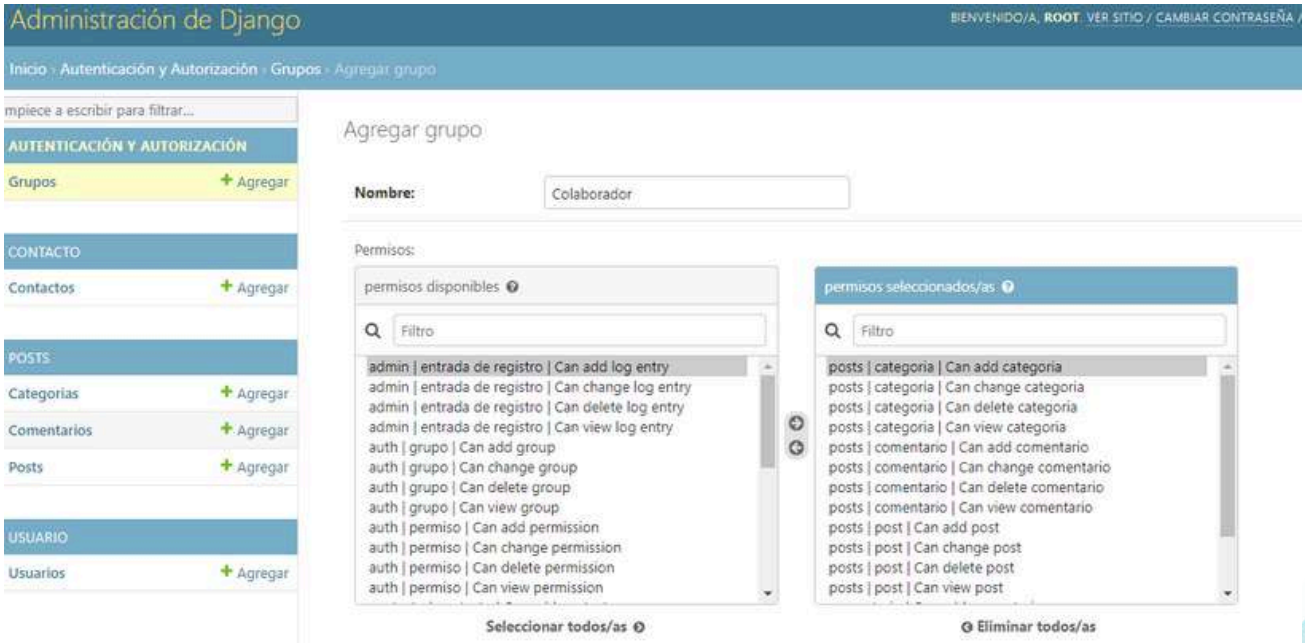
Hasta aquí llegamos con esta parte del apunte. Próximamente te presentaremos filtros y algunos ajustes más...

USUARIO COLABORADOR - USUARIO REGISTRADO

>>> PASOS

Ahora creamos grupos para identificar a los perfiles de usuarios. En este caso vamos a crear los perfiles de usuario Colaborador y Registrado. Esta es una forma de hacerlo sencilla. También puedes crear estos perfiles desde models.py, pero nosotros usaremos esta forma para que sea más sencillo.
Vamos a dirigirnos al admin de Django donde agregaremos un grupo:

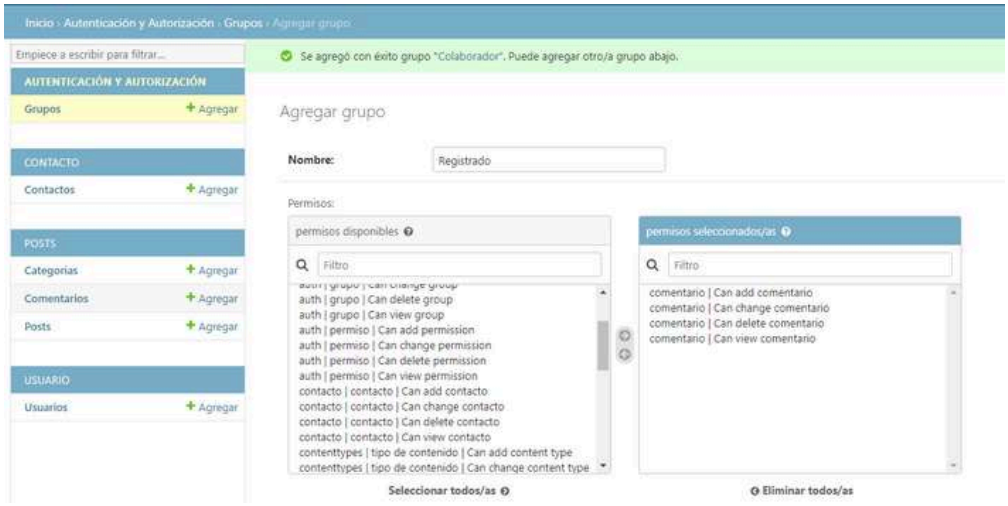
Pondremos el nombre del grupo y le asignaremos los permisos correspondientes. En este caso el usuario Colaborador tendrá todos los permisos para "comentario", "post", "categoria".



USUARIO COLABORADOR - USUARIO REGISTRADO

>>> PASOS

Luego crearemos el usuario Registrado el cual tendrá permisos para "comentario".



Y, haremos una modificación en la view.py de usuario para que cada usuario nuevo que se registre, tendrá asignado por defecto el perfil de usuario "Registrado", ya que a los usuarios "Colaborador" nosotros como "SuperUsuario" deberemos darle el acceso a ser "Colaborador".

apps > usuario > views.py > ...

```
from .forms import RegistroUsuarioForm
from django.contrib.auth.views import LoginView, LogoutView
from django.views.generic import CreateView
from django.contrib import messages
from django.shortcuts import redirect
from django.urls import reverse
from django.contrib.auth.views import PasswordResetView
from django.contrib.auth.views import PasswordResetDoneView
from django.contrib.auth.models import Group

# Create your views here.

class RegistrarUsuario(CreateView):
    template_name = 'registration/registrar.html'
    form_class = RegistroUsuarioForm

    def form_valid(self, form):
        response = super().form_valid(form)
        messages.success(self.request, 'Registro exitoso. Por favor, inicia sesión.')
        group = Group.objects.get(name='Registrado')
        self.object.groups.add(group)
        return redirect('apps.usuario:registrar')
```

USUARIO COLABORADOR - USUARIO REGISTRADO

>>> PASOS

Probamos haciendo un registro de usuario desde nuestra web y luego vamos al admin de Django para verificar que está funcionando correctamente.

En este caso hicimos un registro de un nuevo usuario desde la web llamado "Registrado" y luego fuimos al admin de Django y nos dirigimos al usuario que se creó donde podemos ver que efectivamente, en la casilla de grupos, está marcado como "Registrado".

Y si nos dirigimos al post que tenemos verificamos que ya podemos realizar un comentario como usuario registrado:

USUARIO COLABORADOR - USUARIO REGISTRADO

>>> PASOS

Lo siguiente es crear el menú y accesos para cada usuario. Para ello deberemos armar templates donde el usuario Colaborador podrá crear "posts", modificarlos o eliminarlos, además podrá ver la lista de usuarios que hay registrados para eliminarlos si es necesario y, podrá realizar comentarios sobre los posts, modificar o eliminarlos tanto a los propios, y, los usuarios registrados podrán realizar comentarios en los posts (como lo acabamos de ver) además de modificar o eliminar sus propios comentarios.
Primero vamos a registrar un usuario "colaborador" y darle el acceso como "Colaborador"

Guardamos los cambios y comenzamos con las views para este usuario.
Aclaramos nuevamente, esta es solo una forma de hacerlo, es solo un ejemplo para poder guiare pero tu podrás adaptar a tu código como lo deses.

Aclarado esto, lo que haremos para que este usuario colaborador es darle un botón de acceso para la creación de "posts" desde la sección de "posts", donde cuando este usuario habra los posts, le aparezca un botón para crear posts, sin embargo, para el usuario Registrado no va a pasar lo mismo, este solo tendrá acceso a realizar comentarios.

Quedando algo similar a esto.

Lo primero será crear un formulario para realizar esta acción.
Este formulario, en este caso lo hacemos desde la app "post" pero tranquilamente se puede hacer desde la app "usuario".

USUARIO COLABORADOR - USUARIO REGISTRADO

Para este formulario, además de la creación de post, deberemos crear otro para agregar categorías:

```
blog > apps > posts > forms.py > ...
1   from django import forms
2   from .models import Comentario, Post, Categoria
3
4
5   class ComentarioForm(forms.ModelForm):
6       class Meta:
7           model = Comentario
8           fields = ['texto']
9
10  class CrearPostForm(forms.ModelForm):
11      class Meta:
12          model = Post
13          fields = '__all__'
14
15  class NuevaCategoriaForm(forms.ModelForm):
16      class Meta:
17          model = Categoria
18          fields = '__all__'
19
```

Como vamos a usar todos los campos de estos modelos, usamos la cláusula '__all__' (recuerda que sieimpre debemos importar lo que vamos a usar, en este caso los modelos Post y Categoría).
Un vez realizado esto, debemos crear la vista para que podamos usar estos formularios y para ello necesitaremos importar el modelo Categoria y los formularios CrearPostForm, NuevaCategoriaForm. Además, utilizaremos la función reverse_lazy, por lo que también debemos importarla.

Para la creación de los posts y categorias vamos a heredar de la vista CreateView.

```
apps > posts > views.py > ...
38                  context = self.get_context_data(**kwargs)
39                  context['form'] = form
40                  return self.render_to_response(context)
41
42  class PostCreateView(CreateView):
43      model = Post
44      form_class = CrearPostForm
45      template_name = 'posts/crear_post.html'
46      success_url = reverse_lazy('apps.posts:posts')
47
48  class CategoriaCreateView(CreateView):
49      model = Categoria
50      form_class = NuevaCategoriaForm
51      template_name = 'posts/crear_categoria.html'
52
53      def get_success_url(self):
54          next_url = self.request.GET.get('next')
55          if next_url:
56              return next_url
57          else:
58              return reverse_lazy('apps.posts:post_create')
59
60  class
```

Para las categorías incluímos la función get_success_url, para que nos vuelva a redigir a la página en la que estábamos una vez creada una nueva categoría.

Además, puedes ver que tanto para Post como para Categoria declaramos los html que vamos a usar por lo que vamos a crearlos dentro de la carpeta templates/posts, sin embargo, para seguir el mismo orden, crearemos las urls primero.

USUARIO COLABORADOR - USUARIO REGISTRADO

```python
# apps > posts > urls.py > ...
from django.urls import path
from .views import PostListView, PostDetailView, PostCreateView, CategoriaCreateView

app_name = 'apps.posts'

urlpatterns = [
    path('posts/', PostListView.as_view(), name='posts'),
    path("posts/<int:id>/", PostDetailView.as_view(), name="post_individual"),
    path('post/', PostCreateView.as_view(), name='crear_post'),
    path('post/categoria', CategoriaCreateView.as_view(), name='crear_categoria'),
]
```

Ahora crearemos los templates de crear_post.html y crear_categoria.html:

```html
<!-- templates > posts > crear_post.html > ... -->
{% extends 'base.html' %}

{% block contenido %}

<center>
    <div class="container-fluid" style="margin: 200px;">
    <div class="container-fluid" style="margin-top: 300px;">
        <a id="boton_post" href="{% url 'apps.posts:crear_categoria' %}">Crear nueva categoria</a>
    </div>

    <h1>Crear Post</h1>
    <form method="POST" enctype="multipart/form-data">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Guardar">
    </form>

</div>
</center>
{% endblock %}
```

```html
<!-- templates > posts > crear_categoria.html > ... -->
{% extends 'base.html' %}

{% block contenido %}

<center>
<div class="container-fluid" style="margin: 200px;">
    <h1>Agregar nueva categoria</h1>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Agregar">
    </form>
</div>
</center>
{% endblock %}
```

Ahora incluiremos el botón para acceder desde post.html:

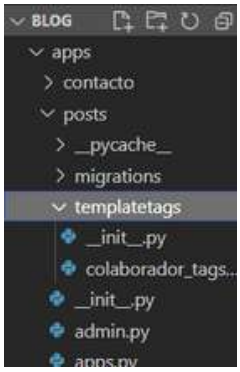
```html
<!-- templates > posts > posts.html > ... -->
{% extends 'base.html' %}
{% load static %}

{% block contenido %}

<div class="container-fluid" style="margin: 200px;">

<div class="container-fluid" style="margin-top: 300px;">
    <a id="boton_post" href="{% url 'apps.posts:crear_post' %}">Crear nuevo post</a>
</div>

    {% for i in posts %}
        <br>
```

Ejecutamos el servidor y probamos.

USUARIO COLABORADOR - USUARIO REGISTRADO

Si lo has probado, tendrás un resultado similar al nuestro, sin embargo, también te habrás dado cuenta que el acceso está abiero para todos, y para que solo sea para usuarios colaboradores o super usuario, tendremos que crear tags y limitar el acceso (recuerda que esto es solo una forma de hacerlo para orientarte, puedes elegir otra forma de hacerlo investigando... te volvemos a recalcar una vez más que, en la programación es necesario leer, estudiar e investigar por cuenta propia para nutrirse de más conocimiento y lograr realizar una misma cosa, pero de diversas formas, más simples o más complejas, según lo que se requiera).

Para realizar estos tags, vamos a crear una nueva carpeta llamada "templatetags" dentro de nuestra aplicación. En este caso en la app posts.



Dentro de esta carpeta pondremos un archivo "__init__.py" vacío (es decir, adentro de él no irá nada) para que Django lea si o si esta carpeta y otro archivo "colaborador_tags.py" donde, dentro de él pondremos el filtro necesario para que el botón que acabamos de crear, solo aparezca cuando sea un usuario "Colaborador".

apps > posts > templatetags > colaborador_tags.py > ...

```
1  from django import template
2
3  register = template.Library()
4
5  @register.filter(name='has_group')
6  def has_group(user, group_name):
7      return user.groups.filter(name=group_name).exists()
```

USUARIO COLABORADOR - USUARIO REGISTRADO

Volvemos a la plantilla posts.html y cargamos el tags. Luego, definimos un condicional para mostrar el botón solo si el usuario es "Colaborador" o "super usuario" (sin embargo, super usuario puede estar como no, solo lo pusimos para mostrarte que se pueden poner distintos usuarios a filtrar con el condicional usando "or", e incluso, si necesitas cumplir más condiciones, puedes usar "and" tal cual lo hacemos en Python).

templates > posts > posts.html > ...

```
{% extends 'base.html' %}
{% load static %}
{% load colaborador_tags %}

{% block contenido %}

<div class="container-fluid" style="margin: 200px;">

{% if user.is_superuser or request.user|has_group:"Colaborador"  %}
    <div class="container-fluid" style="margin-top: 300px;">
        <a id="boton_post" href="{% url 'apps.posts:crear_post' %}">Crear nuevo post</a>
    </div>
{% endif %}

    {% for i in posts %}
```

Ahora para probar, si no detuviste el servidor mientras hacías esta modificaciones, detenlo y vuelve a ejecutar, luego lo probamos con los usuarios que tenemos registrados -- "Colaborador1", "Registrado" y "root".

root y Colaborador

Registrado y no registrado

USUARIO COLABORADOR - USUARIO REGISTRADO

Ahora haremos un listado de categorías donde el usuario Colaborador podrá eliminar las categorías que no quiera o modificarlas. Para ellos,y, porque estamos en la carpeta templates/posts, vamos a crear dos plantillas nuevas "categoria_list.html" y "categoria_confirm_delete.html":

```
templates > posts > categoria_list.html > ...
{% extends 'base.html' %}

{% block contenido %}
<center>
<div class="container-fluid" style="margin: 200px;">
    <h1>Categorías</h1>
    <br>
    <ul>
        {% for categoria in categorias %}
            <li>{{ categoria.nombre }}
            (<a style="color: black;"
            href="{% url 'apps.posts:categoria_delete' pk=categoria.pk %}">Eliminar</a>)</li>
        {% empty %}
            <li>No hay categorías</li>
        {% endfor %}
    </ul>
</div>
</center>
{% endblock %}
```

```
templates > posts > categoria_confirm_delete.html > ...
{% extends 'base.html' %}

{% block contenido %}
<center>
<div class="container-fluid" style="margin: 300px;">
    <h1>Eliminar Categoría</h1>
    <p>¿Estás seguro de que quieres eliminar la categoría "{{ object.nombre }}"?</p>
    <form method="post">
        {% csrf_token %}
        <input type="submit" value="Sí, eliminar">
        <a style="color: black;" href="{% url 'apps.posts:categoria_list' %}">No, cancelar</a>
    </form>
</div>
</center>
{% endblock %}
```

Donde en el template de la lista de categorías pondremos un botón de borrar las categorías en el cual la sintaxis genera una URL para la vista "categoria_delete" (que ahora crearemos) de nuestra app, pasando el valor de "categoria.pk" como argumento.
Ahora iremos a crear las vistas, donde usaremos vistas que nos provee Django (vamos a importar la vista DeleteView, al lado de vista DetailView):

```
apps > posts > views.py > ...
from .models import Post, Comentario, Categoria
from .forms import ComentarioForm, CrearPostForm, NuevaCategoriaForm
from django.views.generic import ListView, DetailView, DeleteView
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic.edit import CreateView
from django.shortcuts import redirect
from django.urls import reverse_lazy


class PostListView(ListView):...

class PostDetailView(DetailView):...

class CrearPostView(LoginRequiredMixin, CreateView):...

class CrearCategoriaView(LoginRequiredMixin, CreateView):...

class CategoriaListView(ListView):
    model = Categoria
    template_name = 'posts/categoria_list.html'
    context_object_name = 'categorias'

class CategoriaDeleteView(DeleteView):
    model = Categoria
    template_name = 'posts/categoria_confirm_delete.html'
    success_url = reverse_lazy('apps.posts:categoria_list')
```

Notarás que prácticamente venimos usando código muy similar para estas nuevas vistas que estamos creando. Esto es gracias a que seguimos reutilizando las vistas genéricas que nos provee Django.

Una vez creadas nuestras vistas, creamos las urls:

USUARIO COLABORADOR - USUARIO REGISTRADO

apps > posts > urls.py > ...

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

app_name = 'apps.posts'

urlpatterns = [
    path('posts/', PostListView.as_view(), name='posts'),
    path("posts/<int:id>/", PostDetailView.as_view(), name="post_individual"),
    path('post/', PostCreateView.as_view(), name='crear_post'),
    path('post/categoria', CategoriaCreateView.as_view(), name='crear_categoria'),
    path('categoria/', CategoriaListView.as_view(), name='categoria_list'),
    path('categoria/<int:pk>/delete/', CategoriaDeleteView.as_view(), name='categoria_delete'),
]
```

Cambiamos por el * para traer todo porque la lista de vistas se hacía muy larga

Volvemos a nuestra plantilla de "crear_categoria" para insertar un acceso a la lista de categorías:

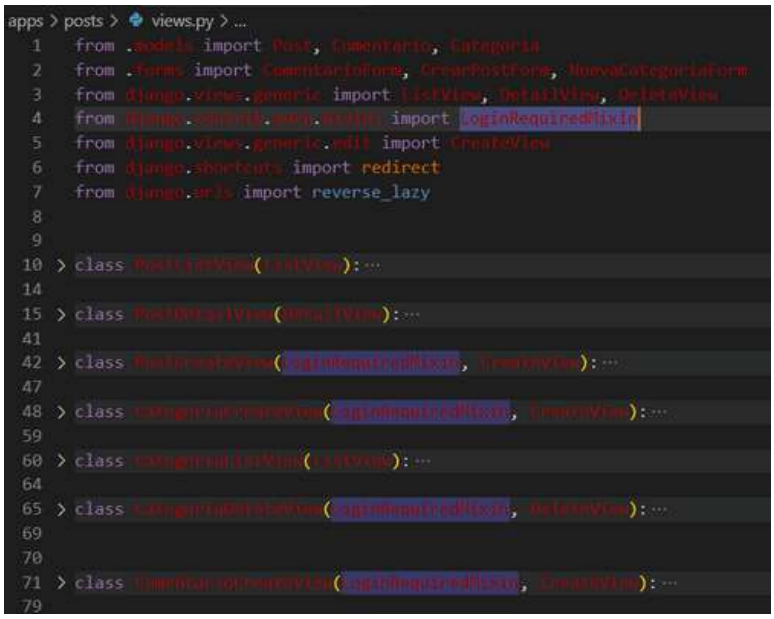
templates > posts > crear_categoria.html > ...

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

{% block contenido %}

<center>
<div class="container-fluid" style="margin: 200px;">
    <h1>Agregar nueva categoría</h1>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Agregar">
    </form>
    <div class="container-fluid" style="margin-top: 300px;">
        <a id="boton_post"
        href="{% url 'apps.posts:categoria_list' %}">Listar todas las categorías</a>
    </div>
</div>
</center>
{% endblock %}
```

Lo probamos:

USUARIO COLABORADOR - USUARIO REGISTRADO

Ahora, si llegaras a deslogueart e ingresas la en la dirección http, por ejemplo "http://127.0.0.1:8000/post/", verás que se puede acceder sin estar logueado, por lo que esto aún sigue liberado para el acceso de cualquier usuario, así que vamos a cerrar para solo los usuarios colaboradores y super usuario.
Vamos a usar "LoginRequiredMixin" como lo hicimos con comentarios, pero también puedes usar decoradores.



Lo único que hacemos es agregar la herencia de "LoginRequiredMixin" a las vistas a restringir, por lo que si ahora lo pruebas nuevamente, de poner la dirección /post (luego de guardar los cambios), no tendrás acceso.

De todas formas, haremos más adeltante, otras modificaciónes para que los usuarios Registrados no puedan acceder tampoco si llegan a poner la dirección en el navegador (pruébalos y verás que si estás logueado como usuario Registrado, podrás de igual manera, crear un post).

También a modo de prueba, podríamos tratar el error 404 de "Page not found", de esta manera, seguiremos organizando nuestro proyecto para lo que es el deploy final al servidor.
Así que con una pequeña pausa, a lo que venimos haciendo, vamos a hacer una página personalizada para "Page not found".
Para esto vamos a realizar las siguientes configuraciones, dirigiendonos primero que nada a settings.py:

PAGE NOT FOUND - 404

```
blog > blog > configuraciones > settings.py > ...
25  # SECURITY WARNING: don't run with debug turned on in production!
26  DEBUG = True
27
28  ALLOWED_HOSTS = []
```

Llevamos estas configuraciones a local.py

```
blog > configuraciones > local.py > ...
1  from .settings import *
2
3  DEBUG = False
4
5  ALLOWED_HOSTS = ['localhost', '127.0.0.1']
6
7  DATABASES = {
8      'default': {
```

Donde para tratar la página 404, de manera local, y, a modo de prueba ponemos el DEBUG en False y en ALLOWED_HOST la configuración del servidor con el que estamos trabajando.
Estas configuraciones también las pondremos en prod.py ya que nos servirá para el deploy del proyecto al final.

```
blog > configuraciones > prod.py > ...
1  from .settings import *
2
3  DEBUG = False
4
5  ALLOWED_HOSTS = []
```

En prod.py si dejaremos completamente en False al DEBUG, ya que si lo dejamos en True como lo tenemos en este momento en modo local, es un gran riesgo para la seguridad de nuestro sitio. Es como dejar la puerta abierta de nuestra casa en una zona con problemas de seguridad.
En local.py por el momento dejaremos en False a DEBUG, solo para mostrar la personalización de la página 404, pero, luego la volveremos a True para seguir trabajando correctamente.
Creareamos una vista personalizada para el template 404. Para ellos vamos a dirigirnos a nuestra views principal y crearemos la siguiente vista:

```
blog > views.py > ...
1  from django.shortcuts import render
2  from django.http import HttpResponseNotFound
3
4  def index(request):
5      return render(request, 'index.html')
6
7  def pagina_404(request, exception):
8      return HttpResponseNotFound('<h1>Página no encontrada</h1>')
```

Luego generaremos la url, de nuestra url principal, para asígnar la función que creamos en la view.

PAGE NOT FOUND - 404

```python
blog > urls.py > ...
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path, include
from . views import index, pagina_404
from django.conf import settings
from django.conf.urls.static import static
from django.contrib.staticfiles.urls import staticfiles_urlpatterns

handler404 = pagina_404

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', index, name='index'),
    path('', include('apps.posts.urls')),
    path('', include('apps.contacto.urls')),
    path('', include('apps.usuario.urls')),

]+static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
urlpatterns += staticfiles_urlpatterns()
urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Asignamos la función de vista para el template 404 al atributo handler404. Guardamos todos los cambios y actualizamos la página en la que estábamos.

Esta es una simple muestra de cómo puedes tratar los errores que puede tener la página y también es una forma de no dejar "abierta" alguna puerta a ataques que pueda recibir nuestra página.
Para seguir trabajando bien y que Django no brinde los errores de manera correcta pudiendo visualizar que está ocurriendo si tenémos errores, vamos a activar nuevamente el DEBUG en True.

```python
blog > configuraciones > local.py > ...
1   from .settings import *
2
3   DEBUG = True
4
```

Y proseguiremos con lo próximo que sería la modificación de posts para nuestro usuario Colaborador.

USUARIO COLABORADOR - USUARIO REGISTRADO

Para realizar la modificación de algún post que se requiera siendo usuario Colaborador, vamos a utilizar otra vista que nos brinda Django. UpdateView.
Nos dirigimos a nuestra view.py de la app posts e importamos la vista UpdateView. Luego creamos la nueva vista para la modificación de los posts:

apps > posts > views.py > ...

```
from .forms import CrearPostForm, CrearCategoriaForm
from django.views.generic import ListView, DetailView, CreateView, UpdateView
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic.edit import CreateView
from django.shortcuts import redirect
from django.urls import reverse_lazy


class PostListView(ListView):...

class PostDetailView(DetailView):...

class PostCreateView(LoginRequiredMixin, CreateView):...

class CategoriaCreateView(LoginRequiredMixin, CreateView):...

class CategoriaListView(ListView):...

class CategoriaDeleteView(LoginRequiredMixin, DeleteView):...

class PostUpdateView(LoginRequiredMixin, UpdateView):
    model = Post
    form_class = CrearPostForm
    template_name = 'posts/modificar_post.html'
    success_url = reverse_lazy('apps.posts:posts')

class CategoriaUpdateView(LoginRequiredMixin, UpdateView):...
```

Donde al igual que las demás vistas, vamos a hacer una herencia de LoginRequiredMixin y de la nueva vista que utilizaremos UpdateView. A su vez, usaremos una plantilla que llamamos "modificar_posts.html".
Crearemos la url y luego el template.

apps > posts > urls.py > ...

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

app_name = 'apps.posts'

urlpatterns = [
    path('posts/', PostListView.as_view(), name='posts'),
    path("posts/<int:id>/", PostDetailView.as_view(), name="post_individual"),
    path('post/', PostCreateView.as_view(), name='crear_post'),
    path('post/categoria', CategoriaCreateView.as_view(), name='crear_categoria'),
    path('categoria/', CategoriaListView.as_view(), name='categoria_list'),
    path('categoria/<int:pk>/delete/', CategoriaDeleteView.as_view(), name='categoria_delete'),
    path('post/<int:pk>/modificar/', PostUpdateView.as_view(), name='post_update'),
]
```

Y ahora creamos el template.

USUARIO COLABORADOR - USUARIO REGISTRADO

```
templates > posts > modificar_post.html > ...
1  {% extends 'base.html' %}
2
3  {% block contenido %}
4
5  <center>
6  <div class="container-fluid" style="margin: 30px;">
7          <a id="boton_post"
8          href="{% url 'apps.posts:crear_categoria' %}?next={% url 'apps.posts:post_update' pk=object.id %}">
9          Crear nueva categoría</a>
10     <br>
11     <h1>Modificar post</h1>
12     <form method="post" enctype="multipart/form-data">
13         {% csrf_token %}
14         {{ form.as_p }}
15         <input type="submit" value="Guardar cambios">
16     </form>
17 </div>
18 </center>
19
20 {% endblock %}
```

En nuestro botón vamos a utilizar el "href" para ir a la url de crear categoría pero también agregaremos un parámetro next a la url utilizando la sintaxis ?next={% url 'apps.posts:post_update' pk=object.id %} para que después de crear la categoría, se redirija al mismo post que estamos modificando.
Lo siguiente del código es igual que la creación de un post.

Este acceso lo agregaremos en "post_individual.html", el cual, se habilitará si el usuario es Colaborador (también puedes agregar que el super usuario tenga el acceso, como te habíamos mostrado en la creación de post).

```
templates > posts > post_individual.html > center > div.container-fluid
1  {% extends 'base.html' %}
2  {% load static %}
3  {% load colaborador_tags %}
4
5  {% block contenido %}
6  <center>
7  <div class="container-fluid" style="margin: 200px;">
8
9      <li>{{ posts.titulo }}</li>
10     <li>{{ posts.subtitulo }}</li>
11     <li>{{ posts.categoria }}</li>
12     <br>
13     <li>{{ posts.texto }}</li>
14     <img src="{{ posts.imagen.url }}" alt="">
15
16     {% if user.is_superuser or request.user|has_group:"Colaborador"  %}
17     <div class="container-fluid" style="margin-top: 300px;">
18         <a id="boton_post" href="{% url 'apps.posts:post_update' pk=posts.id %}">Modificar Post</a>
19     </div>
20     {% endif %}
```

Puedes probar y verás que ya está funcionando la modificación de un post si el usuario es Colaborador o super usuario.

Ahora debemos integrarar un botón para eliminar el post, para ir finalizando con los templates para los posts con acceso Colaborador.

```
11     <li>{{ posts.categoria }}</li>
12     <br>
13     <li>{{ posts.texto }}</li>
14     {% if posts.imagen %}
15     <img src="{{ posts.imagen.url }}" alt="">
16     {% else %}
17     <p>No hay imagen disponible</p>
18     {% endif %}
19     {% if user.is_superuser or request.user|has_group:"Colaborador"  %}
20     <div class="container-fluid" style="margin-top: 300px;">
21         <a id="boton_post" href="{% url 'apps.posts:post_update' pk=posts.id %}">Modificar Post</a>
22     </div>
23     <div class="container-fluid" style="margin-top: 300px;">
24         <a id="boton_post" href="{% url 'apps.posts:post_delete' pk=posts.pk %}">Eliminar Post</a>
25     </div>
26     {% endif %}
27 </div>
```

Por lo que, ya estamos aquí, vamos a insertarlo y realizar una modificación en las imágenes que no realizamos anteriormente. Esto es para que al igual que hicimos con los posts hace varias páginas, de que, si no hay posts, muestre alguna referencia, ya que puede ocurrir también que si no hay imagen, muestre alguna referencia.

USUARIO COLABORADOR - USUARIO REGISTRADO

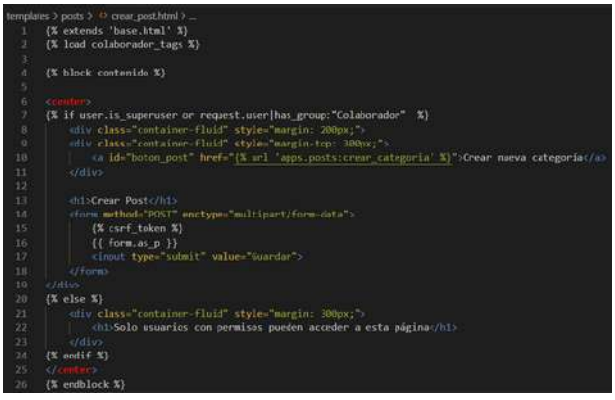
Creamos la view y luego la url.

```
class PostDeleteView(DeleteView):
    model = Post
    template_name = 'posts/eliminar_post.html'
    success_url = reverse_lazy('apps.posts:posts')
```

```
from .views import *

app_name = 'apps.posts'

urlpatterns = [
    path('posts/', PostListView.as_view(), name='posts'),
    path('posts/<int:id>/', PostDetailView.as_view(), name='post_individual'),
    path('post/', PostCreateView.as_view(), name='crear_post'),
    path('post/categoria', CategoriaCreateView.as_view(), name='crear_categoria'),
    path('categoria/', CategoriaListView.as_view(), name='categoria_list'),
    path('categoria/<int:pk>/delete/', CategoriaDeleteView.as_view(), name='categoria_delete'),
    path('post/<int:pk>/modificar/', PostUpdateView.as_view(), name='post_update'),
    path('post/<int:pk>/eliminar/', PostDeleteView.as_view(), name='post_delete'),
]
```

Lo probamos y funciona correctamente. Si te llegara a salir algún error con respecto a la imagen, solo tienes que verificar que la imagen por defecto esté en el lugar correcto mediante el modelo de "post" el cual configuramos en la página 26. Si quieres cambiar de lugar en donde se guarda la imagen por defecto, tienes que cambiarlo desde ahí y hacer las migraciones para que surgan efectos los cambios.

Si has probado introducir la ruta de creación de un post u otras rutas estando como usuario Registrado, verás que aún se puede acceder. Por eso lo que nos queda es realizar las restricciones para que solo los usuarios Colaboradores puedan tener acceso y no los usuarios Registrados. Esto lo tienes que realizar en los templates que el usuario Registrado no puede tener acceso. Por ejemplo:



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

{% block contenido %}

<center>
{% if user.is_superuser or request.user|has_group:"Colaborador"  %}
    <div class="container-fluid" style="margin: 200px;">
    <div class="container-fluid" style="margin-top: 300px;">
        <a id="boton_post" href="{% url 'apps.posts:crear_categoria' %}">Crear nueva categoria</a>
    </div>

    <h1>Crear Post</h1>
    <form method="POST" enctype="multipart/form-data">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Guardar">
    </form>
</div>
{% else %}
    <div class="container-fluid" style="margin: 300px;">
        <h1>Solo usuarios con permisos pueden acceder a esta página</h1>
    </div>
{% endif %}
</center>
{% endblock %}
```

USUARIO COLABORADOR - USUARIO REGISTRADO

Ahora crearemos una lista de usuarios para el Colaborador pueda eliminar usuarios Registrados si así lo desea.
Esta lista la vamos a incluir como un menú desplegable para enseñarte otra forma más que puedes usar para hacer todos tu templates personalizados para los usuarios con permisos especiales.
Vamos a dirigirnos al template base.html donde vamos a agregar el usuario que está activo según el logueo.

templates > base.html > html > body > header > nav > div.nav-content > ul.nav-links > li > a

```
33          <li><a href="{% url 'index' %}">Inicio</a></li>
34          <li><a href="#">Acerca de nosotros</a></li>
35          <li><a href="{% url 'apps.posts:posts' %}">Posts</a></li>
36          <li><a href="{% url 'apps.contacto:contacto' %}">Contacto</a></li>
37          {% if user.is_authenticated %}
38            <li><a href="{% url 'apps.usuario:logout' %}">{{ user.username }}: Logout</a></li>
39          {% else %}
40            <li><a href="{% url 'apps.usuario:login' %}">Login</a></li>
41          {% endif %}
42        </ul>
43      </div>
44    </nav>
```

Donde vamos a agregar una pequeña sintaxtis para que se pueda ver el usuario que está Logueado en este momento
{{ user.username }}

Si lo probamos, ya se puede ver que el nombre del usuario que está logueado sale en el Navbar. Ahora crearemos un menú desplegable:

templates > base.html > html

```
1   {% load static %}
2   {% load colaborador_tags %}
36          <ul class="nav-links">
37            <li><a href="{% url 'index' %}">Inicio</a></li>
38            <li><a href="#">Acerca de nosotros</a></li>
39            <li><a href="{% url 'apps.posts:posts' %}">Posts</a></li>
40            <li><a href="{% url 'apps.contacto:contacto' %}">Contacto</a></li>
41            <div class="dropdown">
42              {% if user.is_authenticated %}
43                <li><a href="{% url 'apps.usuario:logout' %}">{{ user.username }}: Logout</a></li>
44                <div class="dropdown-content">
45                  <table>
46                    <tr>
47                      {% if user.is_superuser or request.user|has_group:"Colaborador"  %}
48                        <td><a href="{% url 'apps.usuario:usuario_list' %}">Lista de usuarios</a></td>
49                      {% endif %}
50                    </tr>
51                    <tr>
52                      <td><a href="#">Lista de categorías</a></td>
53                    </tr>
54                    <tr>
55                      <td><a href="#">Lista de posts</a></td>
56                    </tr>
57                  </table>
58                </div>
59              {% else %}
60                <li><a href="{% url 'apps.usuario:login' %}">Login</a></li>
61              {% endif %}
62            </div>
```

Usamos un poco de css para hacer un menú dropdown (en base.html) al acercar el mouse. Luego vamos a crear la vista para hacer la lista de usuarios. Esto básicamente será como la vista y url que hicimos para las categorías. Te mostramos la view dentro de la app "usuario":

USUARIO COLABORADOR - USUARIO REGISTRADO

```python
# apps > usuario > views.py > ...
class UsuarioListView(LoginRequiredMixin, ListView):
    model = Usuario
    template_name = 'usuario/usuario_list.html'
    context_object_name = 'usuarios'

    def get_queryset(self):
        queryset = super().get_queryset()
        queryset = queryset.exclude(is_superuser=True)
        return queryset

class UsuarioDeleteView(LoginRequiredMixin, DeleteView):
    model = Usuario
    template_name = 'usuario/eliminar_usuario.html'
    success_url = reverse_lazy('apps.usuario:usuario_list')
```

En la vista de "UsuarioListView" usamos un "queryset" para no mostrar al super usuario en la lista así, ante cualquier cosa, un usuario Colaborador no podrá eliminarnos.
Si lo probamos, con estas vistas ya podemos eliminar los usuarios, sin embargo, también vamos a necesitar eliminar comentarios o posts en caso que se requiera. Por lo que usando una función sobre la vista del borrado de usuario, haremos las consultas pertinentes para eliminar o no los posts o comentarios del usuario.

```python
# apps > usuario > urls.py > ...
from django.urls import path
from . import views
from .views import *
from django.contrib.auth import views as auth_views

app_name = 'apps.usuario'

urlpatterns = [
    path('registrar/', views.RegistrarUsuario.as_view(), name='registrar'),
    path('login/', views.LoginUsuario.as_view(), name='login'),
    path('logout/', views.LogoutUsuario.as_view(), name='logout'),
    path('password_reset/', MyPasswordResetView.as_view(), name='password_reset'),
    path('password_reset/done/', MyPasswordResetDoneView.as_view(), name='password_reset_done'),
    path('reset/<uidb64>/<token>/', auth_views.PasswordResetConfirmView.as_view(), name='password_reset_confirm'),
    path('reset/done/', auth_views.PasswordResetCompleteView.as_view(), name='password_reset_complete'),
    path('usuarios/', UsuarioListView.as_view(), name='usuario_list'),
    path('usuario/<int:pk>/eliminar/', UsuarioDeleteView.as_view(), name='usuario_delete'),
]
```

USUARIO COLABORADOR - USUARIO REGISTRADO

Para esto vamos a actulizar la vista de eliminación con la función para borrar posts o comentarios, pero también vamos a dar un contexto para que el mensaje de eliminar posts, aparezca solo para usuarios Colaborador que son los que tienen permisos de realizar posts.

apps > usuario > views.py > ...

```
class UsuarioDeleteView(LoginRequiredMixin, DeleteView):
    model = Usuario
    template_name = 'usuario/eliminar_usuario.html'
    success_url = reverse_lazy('apps.usuario:usuario_list')

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        colaborador_group = Group.objects.get(name='Colaborador')
        es_colaborador = colaborador_group in self.object.groups.all()
        context['es_colaborador'] = es_colaborador
        return context

    def post(self, request, *args, **kwargs):
        eliminar_comentarios = request.POST.get('eliminar_comentarios', False)
        eliminar_posts = request.POST.get('eliminar_posts', False)
        self.object = self.get_object()
        if eliminar_comentarios:
            Comentario.objects.filter(usuario=self.object).delete()

        if eliminar_posts:
            Post.objects.filter(autor=self.object).delete()
        messages.success(request, f'Usuario {self.object.username} eliminado correctamente')
        return self.delete(request, *args, **kwargs)
```

Y como puedes observar, declaramos un nuevo campo para nuestro modelo de Post, por lo que también vamos a actualizar el modelo así podremos tener referencia de quien creó un Post.

apps > posts > models.py > Post

```
class Post(models.Model):
    titulo = models.CharField(max_length=50, null=False)
    subtitulo = models.CharField(max_length=100, null=True, blank=True)
    fecha = models.DateTimeField(auto_now_add=True)
    texto = models.TextField(null=False)
    activo = models.BooleanField(default=True)
    categoria = models.ForeignKey(Categoria, on_delete=models.SET_NULL, null=True, default='Sin categoria')
    imagen = models.ImageField(null=True, blank=True, upload_to='media', default='static/post_default.png')
    publicado = models.DateTimeField(default=timezone.now)
    autor = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE, null=True)

    class Meta:
```

Una vez actualizado el campo, recuerda que debes hacer las migraciones para que los cambios surgan efectos en la base de datos.

USUARIO COLABORADOR - USUARIO REGISTRADO

Por último, actualizamos nuestro template de eliminación y probamos:

templates > usuario > eliminar_usuario.html > ...

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

{% block contenido %}
<center>
{% if user.is_superuser or request.user|has_group:"Colaborador"  %}
<div class="container-fluid" style="margin: 400px;">
    <h1>Eliminar usuario</h1><br>
    <p>¿Estás seguro de que quieres eliminar el usuario "{{ object.username }}"?</p><br>
    <form method="post">
        {% csrf_token %}
        <label for="eliminar_comentarios">¿Desea eliminar también los comentarios del usuario?</label>
        <input type="checkbox" name="eliminar_comentarios" id="eliminar_comentarios">
        <br><br>
        {% if es_colaborador %}
            <label for="eliminar_posts">¿Desea eliminar también los posts del usuario?</label>
            <input type="checkbox" name="eliminar_posts" id="eliminar_posts">
            <br><br>
        {% endif %}

        <input type="submit" value="Sí, eliminar">
        <a style="color: black;" href="{% url 'apps.usuario:usuario_list' %}">No, cancelar</a>
    </form>
</div>
{% else %}
    <div class="container-fluid" style="margin: 300px;">
        <h1>Solo usuarios con permisos pueden acceder a esta página</h1>
    </div>
{% endif %}
</center>
{% endblock %}
```

USUARIO COLABORADOR - USUARIO REGISTRADO

Ahora ya tenemos la eliminación de usuarios y con ellos también se pueden eliminar o no los posts para Colaborador y/o comentarios. Y si el usuario es Registrado, solo veremos la eliminación de comentarios para tildar si se quiere eliminar o no.
Esto nos lleva a poder modificar un comentario o eliminarlo.
Para ello vamos a usar las vistas de modificación y eliminación de Categoria las cuales modificaremos para usar con los comentarios (importaremos reverse, al lado de reverse_lazy):

apps > posts > views.py > ...

```
80
81  class ComentarioCreateView(LoginRequiredMixin, CreateView):...
89
90  class ComentarioUpdateView(LoginRequiredMixin, UpdateView):
91      model = Comentario
92      form_class = ComentarioForm
93      template_name = 'comentario/comentario_form.html'
94
95      def get_success_url(self):
96          next_url = self.request.GET.get('next')
97          if next_url:
98              return next_url
99          else:
100             return reverse('apps.posts:post_individual', args=[self.object.posts.id])
101
102 class ComentarioDeleteView(LoginRequiredMixin, DeleteView):
103     model = Comentario
104     template_name = 'comentario/comentario_confirm_delete.html'
105
106     def get_success_url(self):
107         return reverse('apps.posts:post_individual', args=[self.object.posts.id])
```

Y para las urls:

apps > posts > urls.py > ...

```
1   from django.urls import path
2   from .views import *
3
4   app_name = 'apps.posts'
5
6   urlpatterns = [
7       path('posts/', PostListView.as_view(), name='posts'),
8       path('posts/<int:id>/', PostDetailView.as_view(), name='post_individual'),
9       path('post/', PostCreateView.as_view(), name='crear_post'),
10      path('post/categoria', CategoriaCreateView.as_view(), name='crear_categoria'),
11      path('categoria/', CategoriaListView.as_view(), name='categoria_list'),
12      path('categoria/<int:pk>/delete/', CategoriaDeleteView.as_view(), name='categoria_delete'),
13      path('post/<int:pk>/modificar/', PostUpdateView.as_view(), name='post_update'),
14      path('post/<int:pk>/eliminar/', PostDeleteView.as_view(), name='post_delete'),
15      path('comentario/<int:pk>/editar/', ComentarioUpdateView.as_view(), name='comentario_editar'),
16      path('comentario/<int:pk>/eliminar/', ComentarioDeleteView.as_view(), name='comentario_eliminar'),
17  ]
```

USUARIO COLABORADOR - USUARIO REGISTRADO

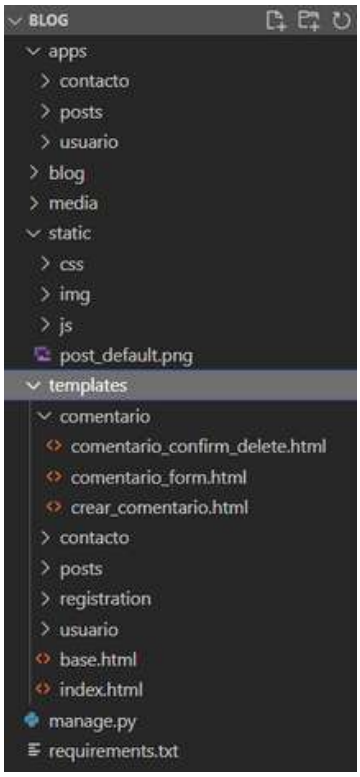
Como puedes ver, estas vistas y urls, son muy similares entre unas y otras ya que al utilizar las vistas genericas de Django, este se encarga de hacer casi todo el trabajo, lo único que nos queda a nosotros, es definir funciones para varias ciertos comportamientos referidos a lo que necesitamos hacer.
Te habrás dado cuenta que también comenzamos a mover los templates de comentarios a una carpeta llamada "comentario" para poder organizar mejor el código, ya que comenzamos a usar más plantillas para comentarios.
Otra cosa que hicimos para enseñarte, es la de usar una incrustación de template dentro de otro template.
Esto lo hicimos en la plantilla de post_individual.html donde teníamos los comentarios, para separar el código. Utilizamos la etiqueta "include" para llamar a un template desde otro template. Quedando así:

templates > posts > post_individual.html > center

```
{% extends 'base.html' %}
{% load static %}
{% load colaborador_tags %}

{% block contenido %}
<center>
<div class="container-fluid" style="margin: 200px;">

    <li>{{ posts.titulo }}</li>
    <li>{{ posts.subtitulo }}</li>
    <li>{{ posts.categoria }}</li>
    <br>
    <li>{{ posts.texto }}</li>
    {% if posts.imagen %}
    <img src="{{ posts.imagen.url }}" alt="">
    {% else %}
    <p>No hay imagen disponible</p>
    {% endif %}
    {% if user.is_superuser or request.user|has_group:"Colaborador"  %}
    <div class="container-fluid" style="margin-top: 300px;">
        <a id="boton_post" href="{% url 'apps.posts:post_update' pk=posts.id %}">Modificar Post</a>
    </div>
    <div class="container-fluid" style="margin-top: 300px;">
        <a id="boton_post" href="{% url 'apps.posts:post_delete' pk=posts.pk %}">Eliminar Post</a>
    </div>
    {% endif %}
</div>

<!-- Aquí seguirán estando los comentarios, pero
    lo separamos en otro template y lo incluímos en este -->

{% include 'comentario/crear_comentario.html' %}

    </center>
{% endblock %}
```

Mediante esta etiqueta, puedes separar el código para hacerlo aún más legible y fácil de trabajar, llamando desde una plantilla a otra plantilla.
Además verás que, para los comentarios, creamos este template "crear_comentario.html" en la carpeta comentario al mismo nivel de las demás carpetas dentro de la carpeta templates. Dentro de crear_comentario se encuentra el mismo código que había aquí, pero un poco modificado para cumplir con lo que requerimos.

USUARIO COLABORADOR - USUARIO REGISTRADO



Te dejamos de muestra la estructura de como quedaron las carpetas. Ahora veremos el template de crear_comentario.html con sus modificaciones:

```
{% load colaborador_tags %}

<div class="container-fluid" style="margin-bottom: 20px;">
    <h2>Comentarios</h2>
    <br><br>
</div>
<div class="container-fluid" style="margin-bottom: 20px;"></div>
    {% for comentario in comentarios %}
        <table>{{ comentario.usuario }} - {{ comentario.fecha }}</table>
        <ul class="w-100 list-unstyled d-flex justify-content-between mb-0">
            <p>{{ comentario.texto }}</p>
            {% if user.is_superuser or comentario.usuario == user %}
                <br><a id="boton_post" href="{% url 'apps.posts:comentario_editar' pk=comentario.pk %}?next={{ request.path }}">Editar</a>
            {% endif %}
            {% if user.is_superuser or comentario.usuario == user or request.user|has_group:"Colaborador" %}
                <br><a id="boton_post" href="{% url 'apps.posts:comentario_eliminar' pk=comentario.pk %}">Eliminar</a>
            {% endif %}
            <br><br>
        </ul>
    {% empty %}
    <li>No hay comentarios - Puedes ser el primero en comentar!</li>
    {% endfor %}
</div>
<a id="comentario"></a>
<div class="container"  style="margin-bottom: 100px;">
<form method="POST" style="margin-bottom: 10px; margin-top: 100px">
    {% csrf_token %}
    {% if user.is_authenticated %}
    <h2>Deja tu comentario</h2>
        <form method="POST">
            {% csrf_token %}
            {{ form.as_p }}
            <input type="submit" value="Comentar">
        </form>
    {% else %}
        <h2>Debes iniciar sesión o registrarte para comentar</h2>
        <a id="boton_post"  href="{% url 'apps.usuario:login' %}?next={{ request.path }}">Iniciar sesión</a>
        <input type="hidden" name="next" value="{{ request.path }}">
    {% endif %}
</form>
</div>
```

En esta primer parte lo que hacemos es agregar los botones de eliminar o modificar comentarios (el resto del código es el mismo que teníamos).

La salvedad que hacemos está relacionada para que el super usuario pueda realizar todo, tanto eliminar o modificar cualquier comentario (esto es opcional ya que el super usuario puede realizar todas las modificaciones que requiera desde el admin de Django).

Luego, para el usuario Colaborador designamos que pueda eliminar cualquier comentario, pero solo pueda modificar los suyos y no los de los usuarios Registrados, aunque con una simple modificación en el template puedes hacer que pueda editar también todos los comentarios, propios y de los Registrados.

Y, los usuarios Registrados podrán eliminar o modificar sus propios comentarios.

Cabe destacar que si hay más de un usuario Colaborador, podrá modificar, además de eliminar, cualquier comentario de otros usuarios Colaboradores, ya que en la sentencia solo estamos llamando al grupo de Colaborador, pero no al Colaborador específico, a diferencia de lo que hacemos con los usuarios Registrados. Esto lo puedes ir manejando en la medida que lo requieras, ya que desde aquí te damos las bases y diversas formas sencillas de hacerlo.

USUARIO COLABORADOR - USUARIO REGISTRADO

Otra cosa para destacar, es que aquí estamos dando accesos desde el propio template usando código Python, sin embargo, esto se puede hacer desde las views.
El otro template que usamos para realizar las modificaciones de los comentarios es el siguiente:

templates > comentario > comentario_form.html > ...

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

{% block contenido %}
<center>
    {% if user.is_superuser or comentario.usuario == user %}
<div class="container-fluid" style="margin: 300px;">
    <h1>Editar comentario</h1>
    <form method="POST" style="margin-bottom: 10px; margin-top: 100px">
        {% csrf_token %}
        {{ form.as_p }}
        <input type="submit" value="Guardar cambios">
    </form>
</div>
{% else %}
    <div class="container-fluid" style="margin: 300px;">
        <h1>Solo usuarios con permisos pueden acceder a esta página</h1>
    </div>
{% endif %}
</center>
{% endblock %}
```

Y el template para eliminar comentarios:

templates > comentario > comentario_confirm_delete.html > ...

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

{% block contenido %}
<center>
{% if user.is_superuser or request.user|has_group:"Colaborador" or comentario.usuario == user %}
    <div class="container-fluid" style="margin: 300px;">
    <h1>Eliminar comentario</h1>
    <p>¿Estás seguro de que deseas eliminar este comentario?</p>
    <form method="POST">
        {% csrf_token %}
        <input type="submit" value="Sí, eliminar">
        <a style="color: black;" href="{{ object.get_absolute_url }}">No, cancelar</a>
    </form>
    </div>
{% else %}
    <div class="container-fluid" style="margin: 300px;">
        <h1>Solo usuarios con permisos pueden acceder a esta página</h1>
    </div>
{% endif %}
</center>
{% endblock %}
```

MOSTRANDO RESULTADOS

Hecho todo lo necesario para poder visualizar, te mostramos los resultados:

Aquí estamos logueados como Colaborador y puedes ver que el Colaborador puede eliminar los comentarios de los usuarios Registrados pero no modificarlos, sin embargo, puede puede modificar y eliminar los comentarios propios y de otros Colaboradores.

Luego hacemos un logue como usuario Registrado

MOSTRANDO RESULTADOS

Hecho todo lo necesario para poder visualizar, te mostramos los resultados:

Lo mismo para otro usuario Registrado, solo puede eliminar y editar su propio comentario.

Y el opcional para el super usuario que puede editar o eliminar cualquier comentario.

AGREGANDO ALGUNOS FILTROS MÁS

Ahora vamos a enlazar los accesos que hicimos para el menú dropdown de base.html de manera tal que el usuario pueda filtrar los posts por categoría e ingresar a la lista de posts por esa categoría, para esto realizamos una pequeña modificación a base.html:

```
templates > base.html > html > body
<header>
  <nav>
    <div class="nav-content">
      <div class="logo">
        <img src="{% static 'img/info.png' %}" id="menu_toggle" style="max-width: 300px; margin: 30px;">
      </div>
      <ul class="nav-links">
        <li><a href="{% url 'index' %}">Inicio</a></li>
        <li><a href="#">Acerca de nosotros</a></li>
        <li><a href="{% url 'apps.posts:posts' %}">Posts</a></li>
        <li><a href="{% url 'apps.contacto:contacto' %}">Contacto</a></li>
        <div class="dropdown">
          {% if user.is_authenticated %}
              <li><a href="{% url 'apps.usuario:logout' %}">{{ user.username }}: Logout</a></li>
          {% else %}
              <li><a href="{% url 'apps.usuario:login' %}">Login</a></li>
          {% endif %}
              <div class="dropdown-content">
                <table>
                  <tr>
                    {% if user.is_superuser or request.user|has_group:"Colaborador"  %}
                      <td><a href="{% url 'apps.usuario:usuario_list' %}">Lista de usuarios</a></td>
                    {% endif %}
                  </tr>
                  <tr>
                      <td><a href="{% url 'apps.posts:categoria_list' %}">Lista de categorías</a></td>
                  </tr>
                </table>
              </div>
        </div>
    </nav>
</header>
```

Con esto, todo usuario que navegue por la web podrá filtrar por categorías los posts. Si el usuario es colaborador, tendrá acceso a la lista de usuarios, y, todos los usuarios podrán loguearse, registrarse o desloguearse.
Además de esto tendremos que hacer una modificación en la vista de post para realizar el filtrado. Como siempre decimos, hay muchas formas de hacerlo. Nosotros crearemos una nueva vista para este filtrado.

AGREGANDO ALGUNOS FILTROS MÁS

Donde tomaremos como filtro las categorías mediante una queryset:

```
apps > posts > views.py > ...
59  class PostsPorCategoriaView(ListView):
60      model = Post
61      template_name = 'posts/posts_por_categoria.html'
62      context_object_name = 'posts'
63
64      def get_queryset(self):
65          return Post.objects.filter(categoria_id=self.kwargs['pk'])
66
```

Verás también que vamos a crear un nuevo template que será posts_por_categoría.html, pero primero configuraremos la url:

```
apps > posts > urls.py > ...
1   from django.urls import path
2   from .views import *
3
4   app_name = 'apps.posts'
5
6   urlpatterns = [
7       path('posts/', PostListView.as_view(), name='posts'),
8       path('posts/<int:id>/', PostDetailView.as_view(), name='post_individual'),
9       path('post/', PostCreateView.as_view(), name='crear_post'),
10      path('post/categoria', CategoriaCreateView.as_view(), name='crear_categoria'),
11      path('categoria/', CategoriaListView.as_view(), name='categoria_list'),
12      path('categoria/<int:pk>/delete/', CategoriaDeleteView.as_view(), name='categoria_delete'),
13      path('post/<int:pk>/modificar/', PostUpdateView.as_view(), name='post_update'),
14      path('post/<int:pk>/eliminar/', PostDeleteView.as_view(), name='post_delete'),
15      path('categoria/<int:pk>/posts/', PostsPorCategoriaView.as_view(), name='posts_por_categoria'),
16      path('comentario/<int:pk>/editar/', ComentarioUpdateView.as_view(), name='comentario_editar'),
17      path('comentario/<int:pk>/eliminar/', ComentarioDeleteView.as_view(), name='comentario_eliminar'),
18  ]
```

Con esto el camino que se seguirá es ingresar a la url de categoria_list desde el menú desplegable dond e se encuetran todas las categorías. Una vez dentro, se elegirá una categoría donde estarán los posts de esa categoría, mediate la url post_por_categoria.html y en el listado de posts que vemos podremos acceder a un post en particular mediante la url que teníamos creada para post_individual.

Como vés, esta es una forma de reutilizar vistas y urls creadas, y siempre se puede hacer de otras formas donde se puede reutilizar aún más o menos.

AGREGANDO ALGUNOS FILTROS MÁS

En post_por_categoria.html mostramos la lista de post ya filtrados y damos acceso a la vista del post individual.

```
templates > posts > <> posts_por_categoria.html > ...
1   {% extends 'base.html' %}
2
3   {% block contenido %}
4   <center>
5   <div class="container-fluid" style="margin: 300px;">
6       <h1>Posts por categoría</h1>
7       <ul>
8           {% for post in posts %}
9               <li><a style="color: ☐black;" href="{% url 'apps.posts:post_individual' id=post.id %}">{{ post.titulo }}</a></li>
10          {% endfor %}
11      </ul>
12  </div>
13  </center>
14  {% endblock %}
```

En post_por_categoria.html mostramos la lista de post ya filtrados y damos acceso a la vista del post individual.
Probamos desde Colaborador1:

Como vés, ya tenemos enlzadas las rutas y accesos. Esto lo puedes integrar en un solo menú en tu proyecto o puedes hacerlo de otra forma. Las posibildades son muchas.

Y ahora para finalizar, te mostraremos los últimos filtros que haremos para ordenar por fecha de más reciente a más antiguo de los posts, o, viceversa, y, por orden alfabético.
Para esto, modificaremos un poco nuestra vista de "PostListView" para lograr estos filtros:

AGREGANDO ALGUNOS FILTROS MÁS

Agregamos un queryset y context_data para sobrescribirlos y hacer los filtros.

apps > posts > views.py > ...

```python
class PostListView(ListView):
    model = Post
    template_name = "posts/posts.html"
    context_object_name = "posts"

    def get_queryset(self):
        queryset = super().get_queryset()
        orden = self.request.GET.get('orden')
        if orden == 'reciente':
            queryset = queryset.order_by('-fecha')
        elif orden == 'antiguo':
            queryset = queryset.order_by('fecha')
        elif orden == 'alfabetico':
            queryset = queryset.order_by('titulo')
        return queryset

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['orden'] = self.request.GET.get('orden', 'reciente')
        return context
```

Luego modificamos la plantilla posts.html para mostrar estas opciones:

AGREGANDO ALGUNOS FILTROS MÁS

templates > posts > posts.html > center > div.container-fluid > h2

```
{% extends 'base.html' %}
{% load static %}
{% load colaborador_tags %}

{% block contenido %}
<center>
<div class="container-fluid" style="margin: 300px;">

{% if user.is_superuser or request.user|has_group:"Colaborador"  %}
    <div class="container-fluid" style="margin-top: 300px;">
        <a id="boton_post" href="{% url 'apps.posts:crear_post' %}">Crear nuevo post</a>
    </div>
{% endif %}

<h2>Ordenar por:</h2>
<ul>
    <a id="boton_post" href="{% url 'apps.posts:posts' %}?orden=reciente">Más reciente</a>
    <a id="boton_post" href="{% url 'apps.posts:posts' %}?orden=antiguo">Más antiguo</a>
    <a id="boton_post" href="{% url 'apps.posts:posts' %}?orden=alfabetico">Orden alfabético</a>
</ul>

    {% for i in posts %}
        <br>
        <li>{{ i.titulo }}</li>
        <li>{{ i.subtitulo }}</li>
        <li>{{ i.categoria }}</li>
        <br>
        <a id="boton_post" href="{% url 'apps.posts:post_individual' i.id %}">
            Ingresá a este Post
        </a>
        {% empty %}
        <h1>No hay registros</h1>
    {% endfor %}
</div>
</center>
{% endblock %}
```

Con esto al ingresar al los posts, podremos ordenar de cualquiera de estas formas. Te mostramos el resultado.

AGREGANDO ALGUNOS FILTROS MÁS

FINALIZACIÓN DEL MANUAL

Con éstas bases, más lo que das en clases y la ayuda con las mentorías, ya tienes un buen conocimiento de Python, Django y SQL.

Recuerda que estas son solo bases para que puedas guiarte e implementar lo aprendido a tu propio proyecto. Siempre puede haber otras formas de resolver cada situación o consigna, y esperamos haberte ayudado con esta guía.

Sigue practicando y buscando diversas formas de resolver cada caso, siempre trata de ser lo más claro posible, busca siempre refactorizar el código para que cada vez sea más legible y fácil de entender. Recuerda que en la mayoría de los casos trabajarás con otros desarrolladores, por lo que mantener el órden, la legibilidad y la facilidad de lectura será lo fundamental para lograr el desarrollo de proyectos eficientes.

Busca ayuda en otros sitios, con otros desarrolladores, con tus profesores o colegas. Hay que reconocer las limitaciones de cada uno y tratar de superarse día a día, practicando y estudiando constantemente ya que esto no se logra de la noche a la mañana.

Desde aquí te deseamos muchos éxitos para tu proyecto final de esta segunda etapa del Informatorio y esperamos que este manual te haya servido de ayuda.