

> Entorno virtual

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 contiene las 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.

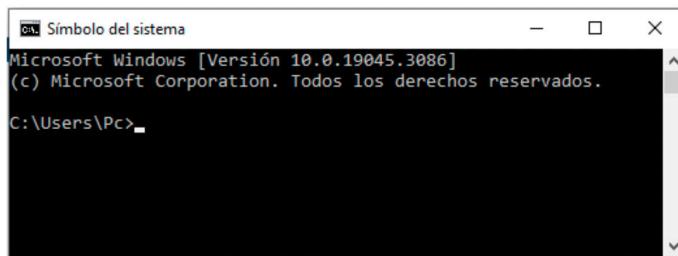
Entonces podemos decir que, un entorno virtual es como tener un taller organizado en sectores, donde cada sector representa un proyecto y contiene las herramientas específicas para ese proyecto. Esto facilita el acceso a las herramientas correctas sin tener que buscar en todas partes, y también proporciona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

PASOS PARA LA CREACIÓN DEL ENTORNO

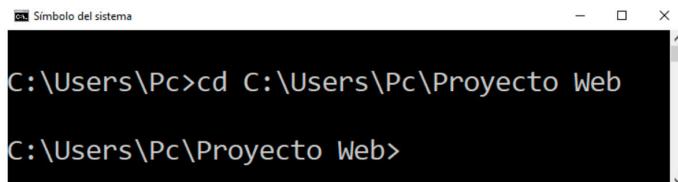
Pasos

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 virutal.

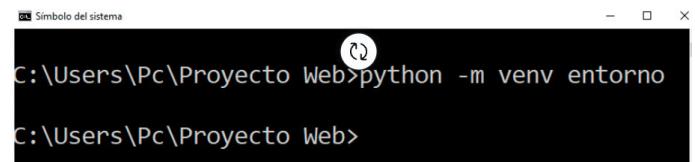


Utiliza el siguiente comando para instalar la paquetería necesaria para crear el entorno:

- 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:

> Instalación de django

Pasos para instalar Django



Pasos

```
Símbolo del sistema
C:\Users\Pc\Proyecto Web>cd entorno
C:\Users\Pc\Proyecto Web\entorno>
```

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:

```
Símbolo del sistema
C:\Users\Pc\Proyecto Web\entorno>cd Scripts
C:\Users\Pc\Proyecto Web\entorno\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:

```
Símbolo del sistema
C:\Users\Pc\Proyecto Web\entorno\Scripts>activate
(entorno) C:\Users\Pc\Proyecto Web\entorno\Scripts>
```

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:

```
Símbolo del sistema
C:\Users\Pc\Proyecto Web\entorno\Scripts>activate
(entorno) C:\Users\Pc\Proyecto Web\entorno\Scripts>cd..
(entorno) C:\Users\Pc\Proyecto Web\entorno>cd..
(entorno) C:\Users\Pc\Proyecto Web>
```

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

> Creación del proyecto

Comenzamos nuestro proyecto



Pasos

```
(entorno) C:\Users\Pc\Proyecto Web>pip install django
Collecting django
  Using cached Django-4.2.2-py3-none-any.whl (8.0 MB)
Collecting asgiref<4,>=3.6.0
  Using cached asgiref-3.7.2-py3-none-any.whl (24 kB)
Collecting sqlparse>=0.3.1
  Using cached sqlparse-0.4.4-py3-none-any.whl (41 kB)
Collecting tzdata
  Using cached tzdata-2023.3-py2.py3-none-any.whl (341 kB)
Installing collected packages: tzdata, sqlparse, asgiref, django
Successfully installed asgiref-3.7.2 django-4.2.2 sqlparse-0.4.4 tzdata-2023.3

[note] A new release of pip available: 22.3.1 -> 23.1.2
[note] To update, run: python.exe -m pip install --upgrade pip
(entorno) C:\Users\Pc\Proyecto Web>
```

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.

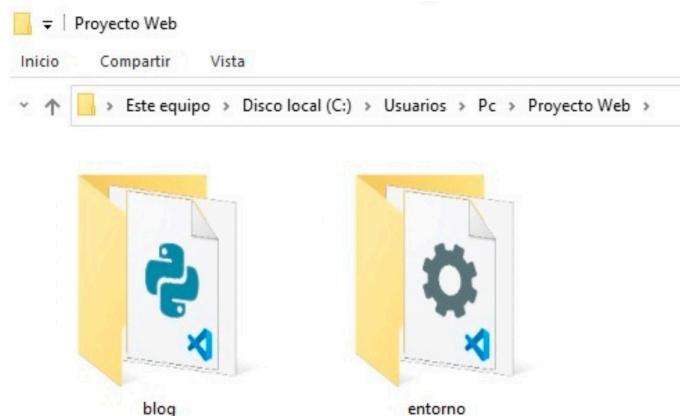
Lo puedes actualizar o seguirlo usando como lo vienes haciendo hasta ahora.***

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):

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

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



> Estructuración de directorios

Estructurando nuestro proyecto



Pasos

Si accedemos a la carpeta 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 fundamentales 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 estuviéramos 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 asíncrona, 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.
- 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 quieras 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 verificar 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
1 archivos       682 bytes
3 dirs   29.880.606.720 bytes libres

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

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

```
(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

[Windows File Explorer]
C:\Users\Pc\Proyecto Web\blog
  - blog
  - manage.py
  - 682 bytes
  - 0 bytes libres

(entorno) C:\Users\Pc\Proyecto Web\blog>mkdir apps static media templates
(entorno) C:\Users\Pc\Proyecto Web\blog>
```

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”.

```
(entorno) C:\Users\Pc\Proyecto Web\blog>mkdir apps static media templates
(entorno) C:\Users\Pc\Proyecto Web\blog>cd blog
(entorno) C:\Users\Pc\Proyecto Web\blog\blog>mkdir configuraciones
(entorno) C:\Users\Pc\Proyecto Web\blog\blog>
```

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

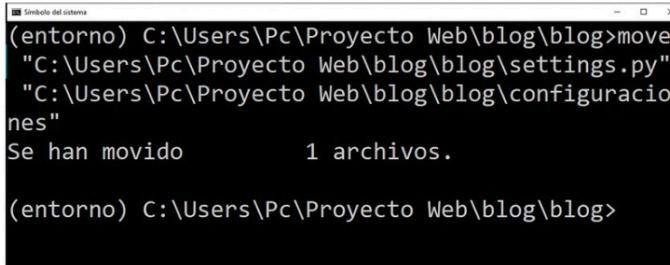
Estando en esta carpeta vamos a mover el archivo settings.py a la carpeta configuraciones que acabamos de crear (puedes hacerlo desde la consola tal cual te mostraremos con el comando move o desde el explorador de Windows cortando y pegando el archivo a la carpeta "configuraciones").



```
(entorno) C:\Users\Pc\Proyecto Web\blog>mkdir apps static media templates  
(entorno) C:\Users\Pc\Proyecto Web\blog>cd blog  
(entorno) C:\Users\Pc\Proyecto Web\blog\blog>mkdir configuraciones  
(entorno) C:\Users\Pc\Proyecto Web\blog\blog>
```

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").



```
(entorno) C:\Users\Pc\Proyecto Web\blog\blog>move  
"C:\Users\Pc\Proyecto Web\blog\blog\settings.py"  
"C:\Users\Pc\Proyecto Web\blog\blog\configuracio  
nes"  
Se han movido 1 archivos.  
(entorno) C:\Users\Pc\Proyecto Web\blog\blog>
```

> Creación de archivos

Creando los archivos local.py y prod.py

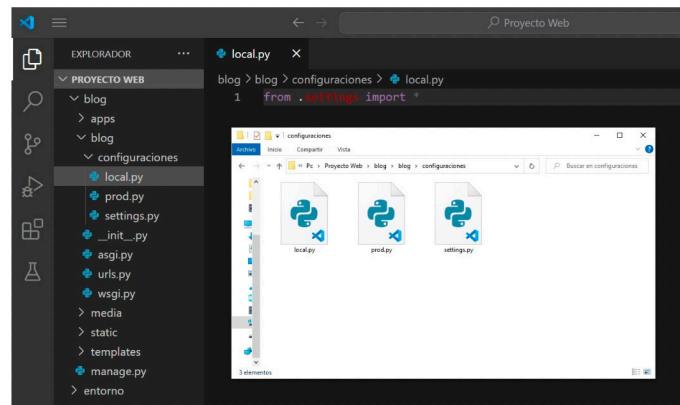


Pasos

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

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

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
57     'BACKEND': 'django.template.backends.django.DjangoTemplates',
58     'DIRS': [os.path.join(os.path.dirname(BASE_DIR), 'templates')],
      '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, botones o imágenes que no cambiaran por todo el recorrido de la web):

```
116 # https://docs.djangoproject.com/en/4.2/howto/static-files/
117
118 STATIC_URL = 'static/'
119 STATICFILES_DIRS = (os.path.join(os.path.dirname(BASE_DIR), 'static'),)
120
121 # Default primary key field type
```

- 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):

```
123
124 DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'
125
126 MEDIA_URL = '/media/'
127 MEDIA_ROOT = os.path.join(os.path.dirname(BASE_DIR), 'media')
128
```

- 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.

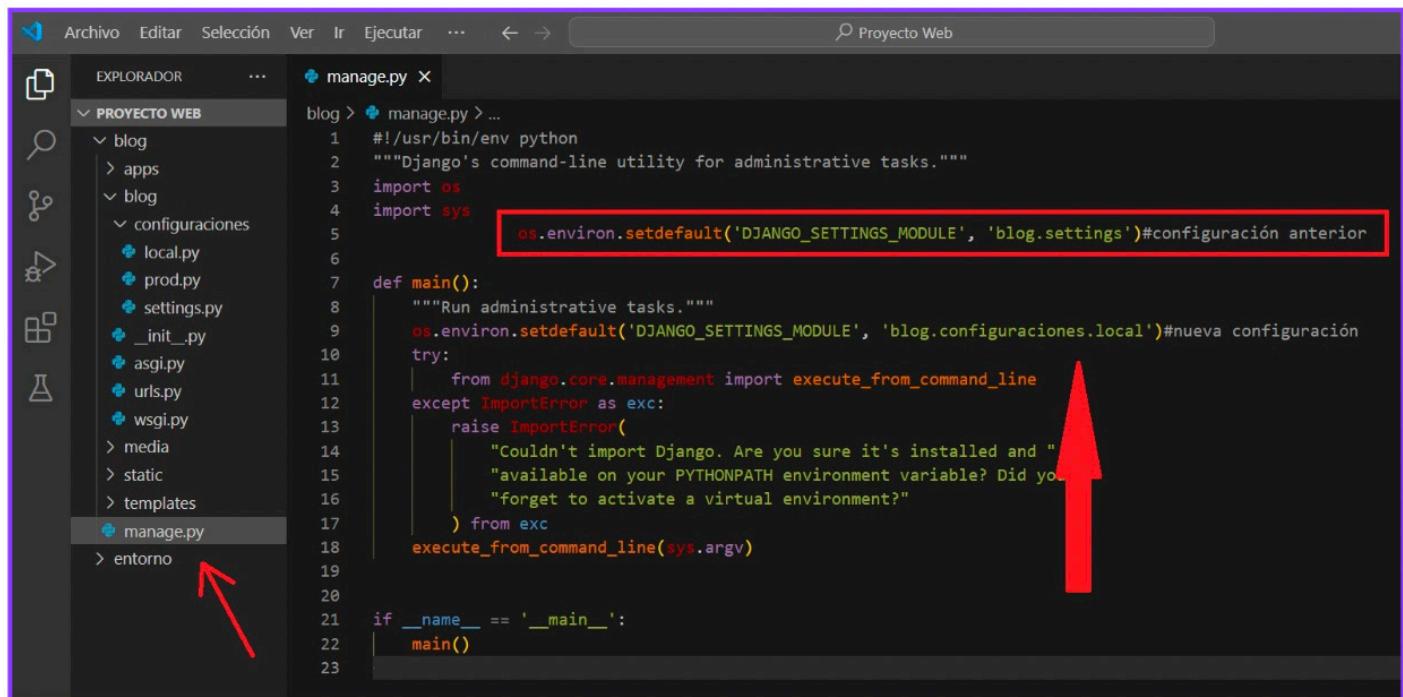
> Modificación de archivos

Configurando el archivo manage.py



Pasos

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



```

    Archivo Editar Selección Ver Ir Ejecutar ... ⌘ ⌘ Proyecto Web
EXPLORADOR ...
PROYECTO WEB
blog > manage.py ...
blog > blog
blog > apps
blog > configuraciones
    local.py
    prod.py
    settings.py
    __init__.py
    asgi.py
    urls.py
    wsgi.py
media
static
templates
manage.py
entorno

manage.py > ...
blog > manage.py > ...
1  #!/usr/bin/env python
2  """Django's command-line utility for administrative tasks."""
3  import os
4  import sys
5  os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'blog.settings')#configuración anterior
6
7  def main():
8      """Run administrative tasks."""
9      os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'blog.configuraciones.local')#nueva configuración
10     try:
11         from django.core.management import execute_from_command_line
12     except ImportError as exc:
13         raise ImportError(
14             "Couldn't import Django. Are you sure it's installed and "
15             "available on your PYTHONPATH environment variable? Did you "
16             "forget to activate a virtual environment?"
17         ) from exc
18     execute_from_command_line(sys.argv)
19
20
21 if __name__ == '__main__':
22     main()

```

Vamos a explicarte que hicimos aquí.

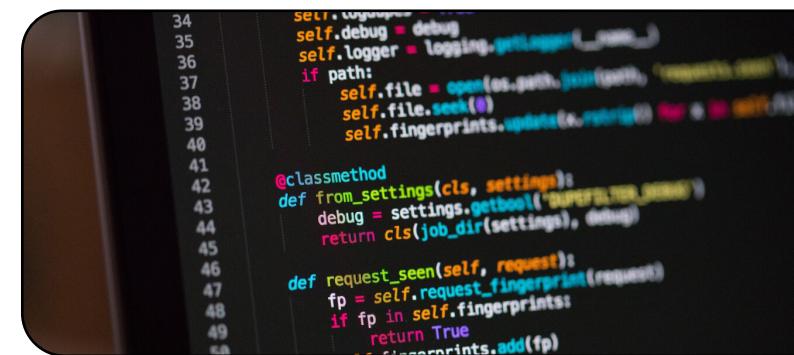
- Como lo mencionamos antes, el archivo `settings.py` tiene la configuración principal de nuestro proyecto, sin embargo, cuando trabajamos de forma local (en nuestra pc), tenemos configuraciones que solo vamos a usar en nuestra pc, pero cuando el proyecto se suba a nuestro repositorio y luego a un servidor, 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 `local.py` las

➤ Corriendo el servidor

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:



```

34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
      self.webpage = webpage
      self.debug = debug
      self.logger = logging.getLogger(__name__)
      if path:
          self.file = open(os.path.join(path, 'request.log'), 'w')
          self.file.seek(0)
          self.fingerprints.update(self._get_fingerprints())
      @classmethod
      def from_settings(cls, settings):
          debug = settings.getbool('DJANGO_DEBUG')
          return cls(job_dir(settings), debug)
      def request_seen(self, request):
          fp = self.request_fingerprint(request)
          if fp in self.fingerprints:
              return True
          self.fingerprints.add(fp)
  
```

`os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'blog.configuraciones.local')` encuentra el archivo `manage.py`.

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

The screenshot shows a Visual Studio Code interface with a terminal window open. The terminal shows the following command sequence:

```
C:\Users\Pc\Proyecto Web\entorno\Scripts>activate #Activamos el entorno
(entorno) C:\Users\Pc\Proyecto Web\entorno\Scripts>cd C:\Users\Pc\Proyecto Web\blog #Volvemos a "blog"
(entorno) C:\Users\Pc\Proyecto Web\blog>python manage.py runserver #Ejecutamos el comando
Watching for file changes with StatReloader
Performing system checks...
System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s):
  admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.

Django version 4.2.2, using settings 'blog.configuraciones.local' #Configuración que realizamos en manage.py
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

A red arrow points from the text "Elegir la opción Command Prompt" to the terminal window. Another red arrow points to the "python" icon in the toolbar above the terminal.

> Página de inicio de django

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), presionando 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.

¡Felicitaciones! 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 queremos en nuestro proyecto. Por ejemplo: al momento de acceder a `empleo.chaco.gob.ar` vemos la página principal de la Subsecretaría 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"

The screenshot shows a Visual Studio Code interface with a file named `views.py` open in the editor. The file contains the following code:

```
from django.shortcuts import render

def index(request):
    return render(request, 'index.html')
```

A red arrow points from the text "views.py" in the file name to the file itself. Another red arrow points to the "views.py" icon in the sidebar.

> Views y urls

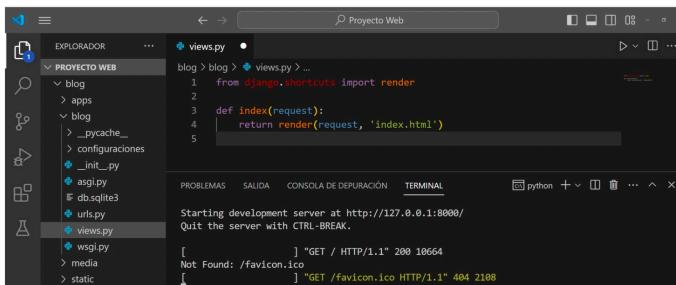
Declarando views y urls



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).

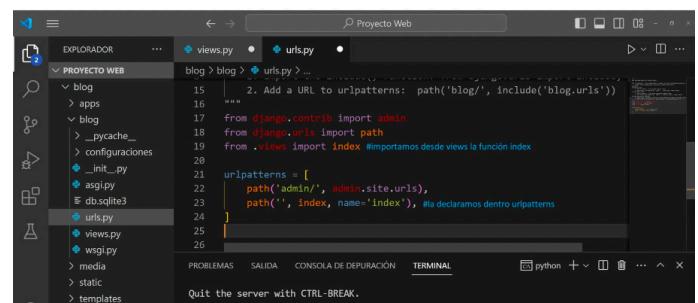
Para poder mostrar esta página inicial, vamos a crear una función en nuestro archivo `views.py`, importando primero que nada una función que permitirá renderizar el archivo html que indiquemos:



```
blog > blog > views.py > ...
1  from django.shortcuts import render
2
3  def index(request):
4      return render(request, 'index.html')
```

```
blog > blog > views.py > ...
1  from django.shortcuts import render
2
3  def index(request):
4      return render(request, 'index.html')
```

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



```
blog > blog > urls.py > ...
15  """
16  """
17  2. Add a URL to urlpatterns: path('blog/', include('blog.urls'))
18  from django.contrib import admin
19  from django.urls import path
20  from .views import index #importamos desde views la función index
21
22  urlpatterns = [
23      path('admin/', admin.site.urls),
24      path('', index, name='index'), #la declaramos dentro urlpatterns
25  ]
26
```

```
19  from .views import index #importamos desde views la función index
20
21  urlpatterns = [
22      path('admin/', admin.site.urls),
23      path('', index, name='index'), #la declaramos dentro urlpatterns
24  ]
```

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 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, incluimos 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 ello 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).

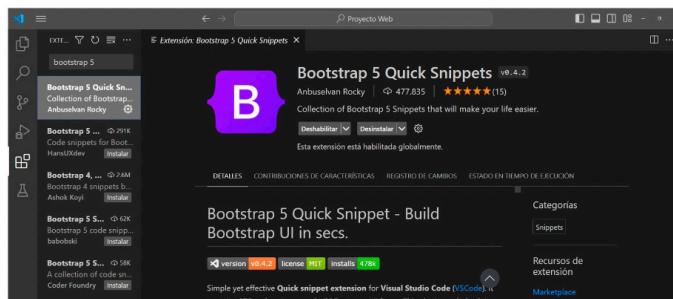
Ahora a fines prácticos cambiaremos el título de la página y el contenido que se verá dentro de la página.

> Plantillas - templates

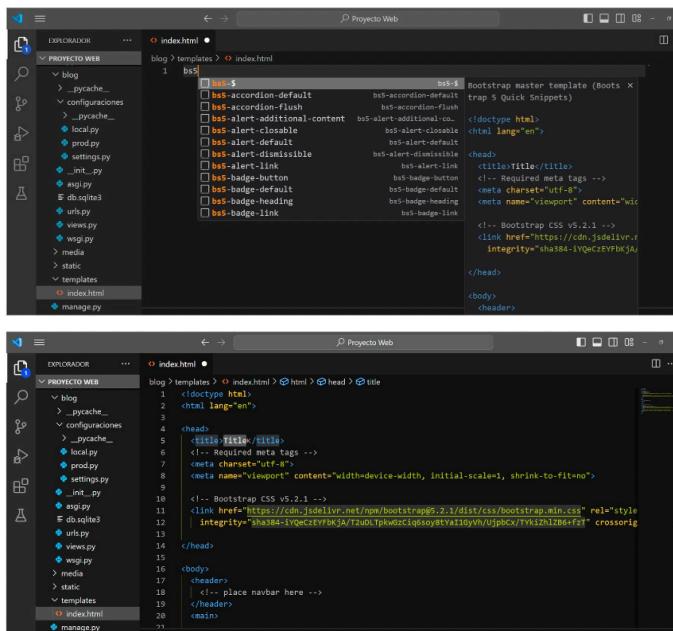
Cambiando nuestra página principal (index)

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 auto-completada 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:



Esta es nuestra página de inicio

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 adelante), 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 mantenimiento 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 avanzando, 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.

> 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 iniciamos 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.
- **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.

```

blog
└── apps
    └── posts
        ├── migrations
        ├── __init__.py
        ├── admin.py
        ├── apps.py
        ├── models.py
        ├── tests.py
        └── views.py
    └── blog

```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL

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

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

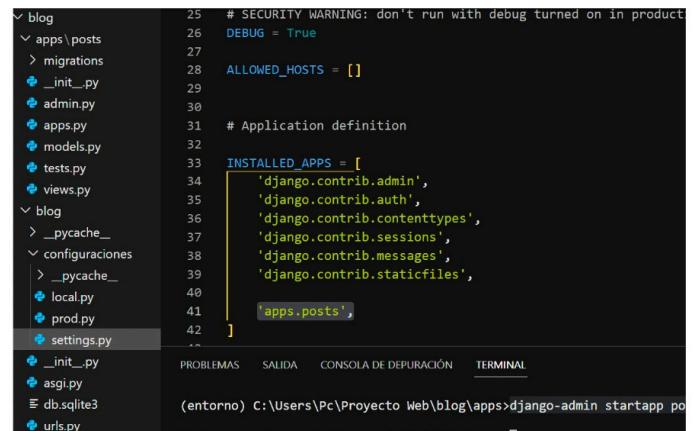
(entorno) C:\Users\Pc\Proyecto Web\blog\apps>

- **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 espacio entre las aplicaciones pre-instaladas y las aplicaciones que creamos.



```

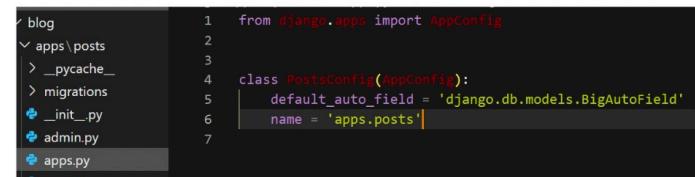
blog
└─ apps\posts
    ├ migrations
    └─ __init__.py
        ├ admin.py
        └─ models.py
            ├ tests.py
            └─ views.py
    └─ __pycache__
        └─ __init__.py
            ├ local.py
            └─ prod.py
                └─ settings.py
                    └─ db.sqlite3
                        └─ urls.py
25     # SECURITY WARNING: don't run with debug turned on in product
26     DEBUG = True
27
28     ALLOWED_HOSTS = []
29
30
31     # Application definition
32
33     INSTALLED_APPS = [
34         'django.contrib.admin',
35         'django.contrib.auth',
36         'django.contrib.contenttypes',
37         'django.contrib.sessions',
38         'django.contrib.messages',
39         'django.contrib.staticfiles',
40
41         'apps.posts',
42     ]

```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL
(entorno) C:\Users\Pc\Proyecto Web\blog\apps>django-admin startapp po

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`’



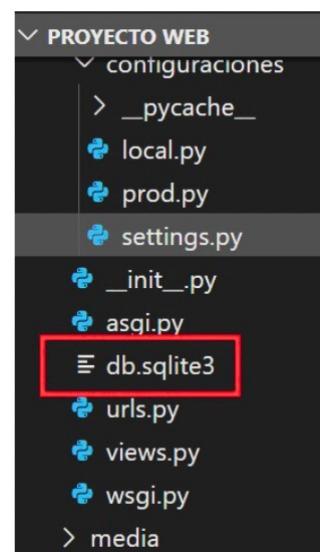
```

blog
└─ apps\posts
    ├ migrations
    └─ __init__.py
        ├ admin.py
        └─ apps.py
            └─ db.sqlite3
                └─ urls.py
1     from django.apps import AppConfig
2
3
4     class PostsConfig(AppConfig):
5         default_auto_field = 'django.db.models.BigAutoField'
6         name = 'apps.posts'
7

```

No olvides siempre guardar los cambios con `ctrl +s`.

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 aparecían 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).

```
System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for apps
>: admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
```

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

```

PROYECTO WEB
= views.py[python-311...]
≡ wsgi.py[python-311...]
✓ configuraciones
> _pycache_
↳ local.py
↳ prod.py
↳ settings.py
↳ __init__.py
↳ asgi.py
≡ db.sqlite3
↳ urls.py
↳ views.py
> ESQUEMA
> LÍNEA DE TIEMPO

PROBLEMAS SALIDA TERMINAL ...
cmd + v ⌂ ⌂ ⌂ ...
```

```

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

C:\Users\Pc\Proyecto Web\blog>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
```

No te preocupes que al escribir makemigrations salga la leyenda “No changes 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 acercamiento 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 visualizarla 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.

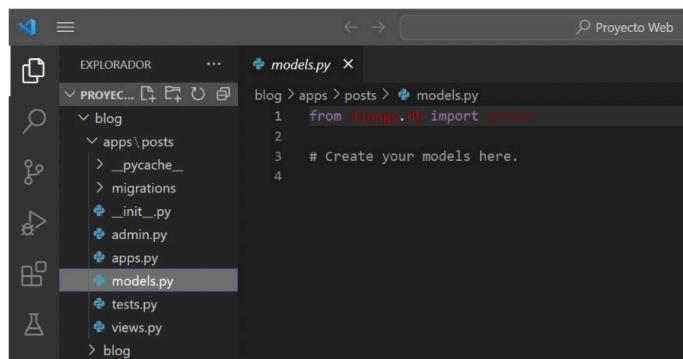
> Modelos

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”:

Ahora crearemos una clase “Categoría” como lo hacíamos en POO, tomando a categoría como un objeto, y si lo vieramos desde SQL lo tomaríamos como una entidad.

```

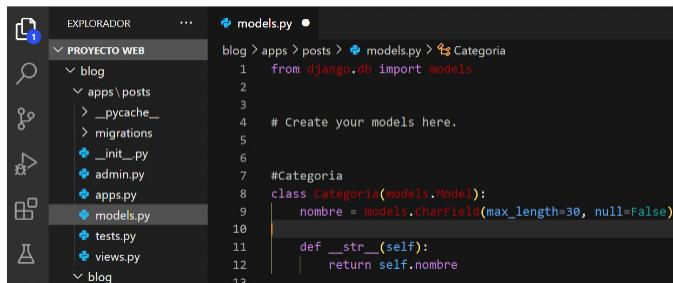
models.py
blog > apps > posts > models.py > Categoria
1   from django.db import models
2
3
4   # Create your models here.
5
6
7   #Categoria

```

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 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.



```

EXPLORADOR
PROYECTO WEB
blog > apps > posts > models.py > Categoria
1  from django.db import models
2
3
4  # Create your models here.
5
6
7  class Categoria(models.Model):
8      nombre = models.CharField(max_length=30, null=False)
9
10     def __str__(self):
11         return self.nombre
12
13

```

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:

```

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      subtitle = 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 categoria')
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 “Categoría” 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.
- **subtitle:** 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 estrictamente 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 a tener, 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.
- **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 vacío, 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 imágenes 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 imágenes que se carguen cuando se cree un post, se subirán a la carpeta "media" ya que éstas imágenes forman la parte dinámica de la web, donde el o los usuarios que realicen la carga de post, irán subiendo o modificando las imágenes conforme se requiera, pero para las imágenes 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

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":

```

14 class Post(models.Model):
15     titulo = models.CharField(max_length=50, null=False)
16     subtítulo = 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     categoría = 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)
23
24     class Meta:
25         ordering = ('-publicado',)
```

```

25
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 eliminan las imágenes asociadas a ese post. De esta forma no saturaremos el servidor con imágenes 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.

```

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

Adicional al método "delete" usaremos los parámetros opcionales "using" y "keep_parents" para controlar 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.

Como usaremos imágenes 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

> Dependencias adicionales

Instalando dependencias adicionales - requirements.txt

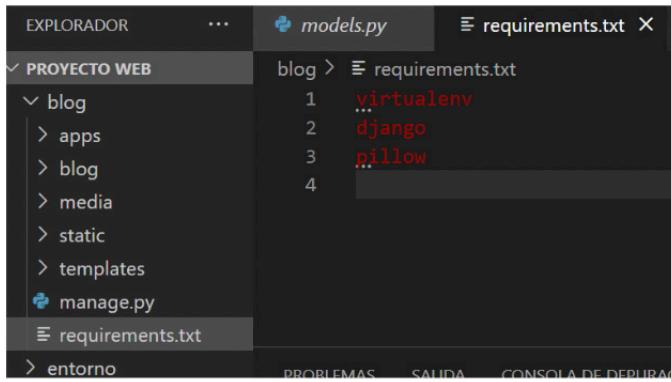


Pasos

repository a Github y alguien quisiera descargarlo y trabajar con él, con solo un comando podría instalar todas las dependencias usadas en el proyecto.

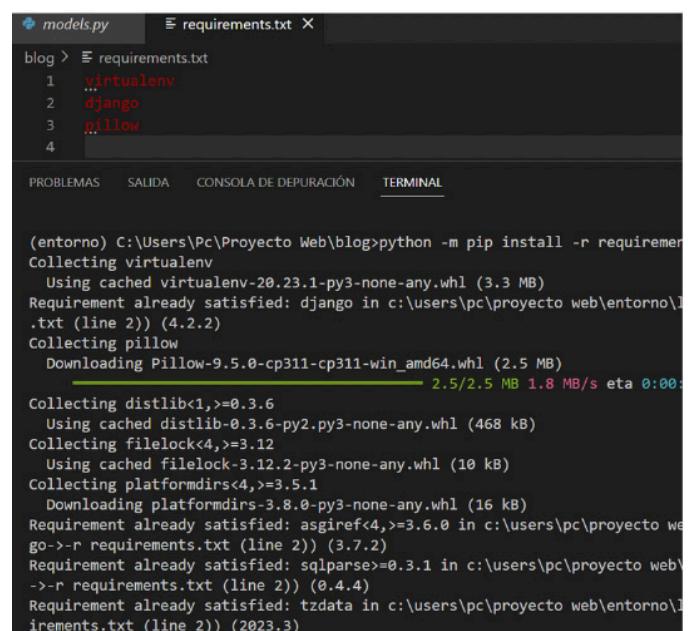
Este archivo lo podremos crear de forma manual o automática. Ahora lo haremos de forma manual, pero luego te daremos el comando para hacerlo de forma automática. Vamos a posicionarnos en la carpeta "blog", donde se encuentra el archivo "manage.py", ahora crearemos un archivo llamado "requirements.txt" (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).



```
models.py requirements.txt
blog > requirements.txt
1 virtualenv
2 django
3 pillow
4
```

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



```
(entorno) C:\Users\Pc\Proyecto Web\blog>python -m pip install -r requirements.txt
Collecting virtualenv
  Using cached virtualenv-20.23.1-py3-none-any.whl (3.3 MB)
Requirement already satisfied: django in c:\users\pc\proyecto web\entorno\django (4.2.2)
Collecting pillow
  Downloading Pillow-9.5.0-cp311-cp311-win_amd64.whl (2.5 MB)
Collecting distlib<1,>=0.3.6
  Using cached distlib-0.3.6-py3-none-any.whl (468 kB)
Collecting filelock<4,>=3.12
  Using cached filelock-3.12.2-py3-none-any.whl (10 kB)
Collecting platformdirs<4,>=3.5.1
  Downloading platformdirs-3.8.0-py3-none-any.whl (16 kB)
Requirement already satisfied: asgiref<4,>=3.6.0 in c:\users\pc\proyecto web\asgi>->r requirements.txt (line 2)) (3.7.2)
Requirement already satisfied: sqlparse=>0.3.1 in c:\users\pc\proyecto web\sqlparse>->r requirements.txt (line 2)) (0.4.4)
Requirement already satisfied: tzdata in c:\users\pc\proyecto web\tzdata>->r requirements.txt (line 2)) (2023.3)
```

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

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 “requirements.txt” ya que si por algún motivo, en una nueva actualización de estas dependencias 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 ocasionar 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.

```
blog > requirements.txt
1 virtualenv==20.23.1
2 django==4.2.2
3 pillow==9.5.0
4

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>
```

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 “requirements.txt”, se cree de forma automática.

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
4

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 dependencias 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.

> Migraciones con nuestro modelo

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 x
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     ]
```

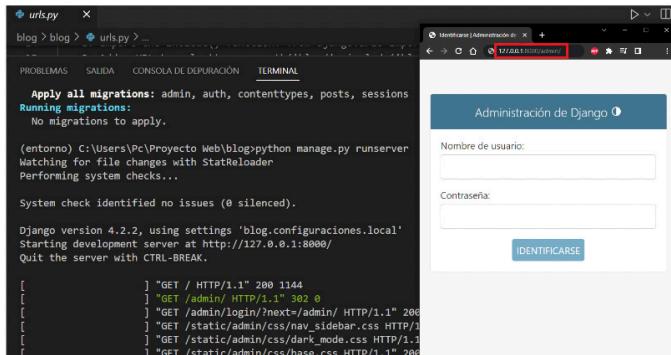
> Admin

Accediendo al administrador de django



Pasos

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



```
(entorno) C:\Users\Pc\Proyecto Web\blog>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...
System check identified no issues (0 silenced).

Django version 4.2.2, using settings 'blog.configuraciones.local'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

[...]
[ ] GET / HTTP/1.1" 200 1144
[ ] GET /admin/ HTTP/1.1" 302 0
[ ] "GET /admin/login/?next=/admin/ HTTP/1.1" 200
[ ] "GET /static/admin/css/nav_sidebar.css HTTP/1.1"
[ ] "GET /static/admin/css/dark_mode.css HTTP/1.1"
[ ] "GET /static/admin/css/base.css HTTP/1.1" 200
```

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/>).

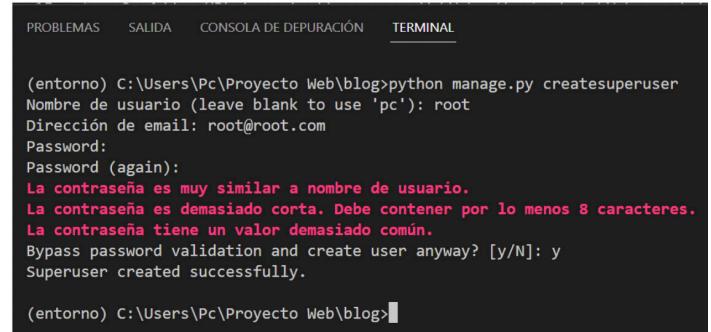
Sin embargo, por más que intentes poner cualquier usuario y contraseña, no podrás acceder, y esto es porque aún no hemos generado nuestro usuario y contraseña para el administrador de Django.

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)

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.

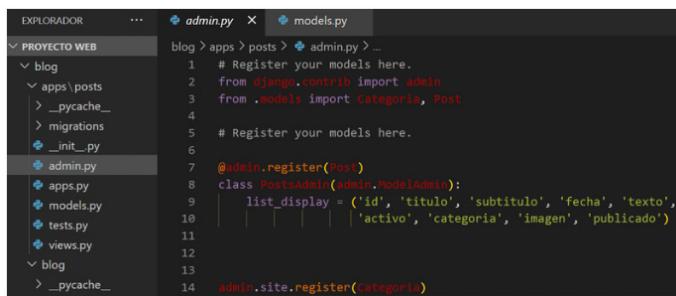


Dentro del sitio tendremos este menú, donde explicaremos cada cosa en la siguiente división del material.

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 visualizarse desde aquí.

Vamos a dirigirnos al archivo admin.py ubicado en la carpeta apps.

Dentro de este archivo, lo completaremos de la siguiente forma (no olvides que siempre debes guardar los cambios con ctrl + s):



```

EXPLORADOR ... admin.py x models.py
PROYECTO WEB ...
blog > apps > posts > admin.py > ...
1 # Register your models here.
2 from django.contrib import admin
3 from .models import Categoria, Post
4
5 # Register your models here.
6
7 @admin.register(Post)
8 class PostsAdmin(admin.ModelAdmin):
9     list_display = ('id', 'titulo', 'subtitulo', 'fecha', 'texto',
10                    'activo', 'categoria', 'imagen', 'publicado')
11
12
13
14 admin.site.register(Categoria)

```

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
4
5 @admin.register(Post)
6 class PostsAdmin(admin.ModelAdmin):
7     list_display = ('id', 'titulo', 'subtitulo', 'fecha', 'texto',
8                    'activo', 'categoria', 'imagen', 'publicado')
9
10
11
12
13
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.
- 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 "Categoria".

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



Administración de sitio

AUTENTICACIÓN Y AUTORIZACIÓN

Grupos		
Usuarios		

POSTS

Categorías		
Posts		

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.