Usando esta clase lo que decimos es que cuando se ejecute nos devolverá un formulario.

from django.contrib.auth.forms import UserCreationForm

Django permite registrar usuarios, pero con el código siguiente nos permitirá reutilizar su clase User para registrar los nuestros

from django.contrib.auth.models import User

Al crear un usuario Django crea las cookies por nosotros importando de la siguiente manera:

from django.contrib.auth import login

Y después en la función usamos el método login() de la siguiente manera:

def signup(request):

    if request.method == "GET":

        form = UserCreationForm()

        return render(request, "signup.html", {"form": form})

    else:

        if request.POST["password1"] == request.POST["password2"]:

*# register user*

            try:

                user = User.objects.create\_user(

                    username=request.POST["username"], password=request.POST["password1"])

                user.save()

                login(request, user)

                return redirect("tasks")

También debemos de importar el método logout() para cerrar sesión. También deberemos de importar el método authenticate que nos permite autenticar el ususario,  
es decir, si es correcto o no.

from django.contrib.auth import login, logout, authenticate

Al igual que hay un UserCreationForm que sirve para crear un usuario y que lo importamos de django.contrib.auth.forms, también tenemos el AuthenticationForm  
que sirve para comprobar si existe un usuario o no.

from django.contrib.auth.forms import UserCreationForm, AuthenticationForm

Cuando en models creemos clases que crean tablas en la base de datos a través del ORM debemos de escribir en el terminal el siguiente código y esto hace que python genere por nosotros la tabla en la base de datos,

PS D:\Datos\CRUDenDjango> python manage.py makemigrations

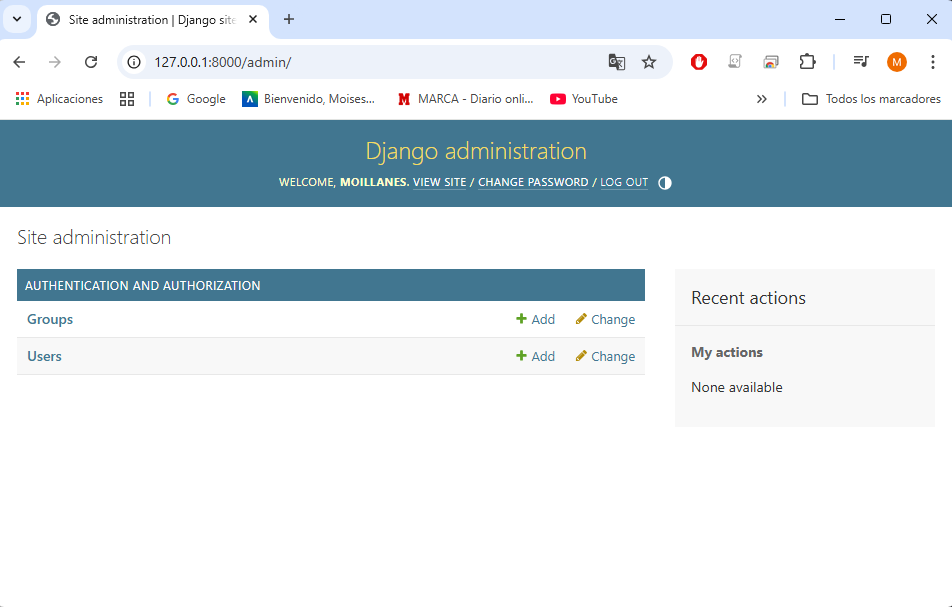
Y para ejecutarla debemos escribir el siguiente código en el terminal:

PS D:\Datos\CRUDenDjango> python manage.py migrate

Para crear un usuario en el panel de administrador y poder gestionar nuestra aplicación desde este panel debemos de aplicar el siguiente código en el terminal:

PS D:\Datos\CRUDenDjango> python manage.py createsuperuser

Este código de arriba nos pedirá un username, email address y escribir dos veces la password para verificarla y entonces se creará nuestro usuario administrador correctamente.



Después le daremos permiso a este panel de administrador para que pueda acceder a crear las tareas desde él y para ello importaremos en el archivo admin.py desde el archivo models la clase Task que tiene los atributos de las tareas y crea las tablas mediante el ORM.

from django.contrib import admin

from .models import Task

*# Register your models here.*

admin.site.register(Task)

Para crear un formulario en html que me pida los atributos de un modelo (clase) crearemos un archivo dentro de nuestra app llamado forms.py en el cual desde django.forms importaremos la clase ModelForm que nos permite poder extenderla. Seguidamente tendremos que importar desde .models las clases que queramos extender o aplicar en los formularios para obtener en él los atributos de esta.

from django.forms import ModelForm

from .models import Task

class TaskForm(ModelForm):

    class Meta:

*#Meta es una clase interna que nos permite configurar el formulario.*

        model = Task

*#model nos indica de qué modelo se va a crear el formulario. En este caso, de Task.*

        fields = ["title", "description", "important"]

Después en views.py importaremos desde .forms la clase que hemos creado para el formulario, en este caso llamada TaskForm para pasársela a la función como parámetro en relación al “form”.

def create\_task(request):

    return render(request, "create\_task.html", {

        "form": TaskForm(),

    })

Para establecer una tarea como completada debemos de importar desde django.utils su método timezone en views y ahora veremos cómo lo aplicamos en la función que convierte una tarea no completada en completada.

def complete\_task(request, task\_id):

    task = get\_object\_or\_404(Task, pk=task\_id, user=request.user)

    if request.method == "POST":

        task.datecompleted = timezone.now()

*#timezone.now() nos devuelve la fecha y hora actual.*

        task.save()

*#Guardamos la tarea después de cambiar y aplicar la fecha y hora actual en el atributo datecompleted de la variable task, que recordemos que por defecto está en nulo, es decir, que no tiene ni fecha ni hora.*

        return redirect("tasks")

Para proteger nuestras rutas importaremos un decorador especial llamado login\_required y que lo podemos colocar en cada función y de esta forma protegemos a la función que se lo pongamos. Aquí vemos un ejemplo:

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

***@login\_required***

def delete\_task(request, task\_id):

    task = get\_object\_or\_404(Task, pk=task\_id, user=request.user)

    if request.method == "POST":

        task.delete()

        return redirect("tasks")

Si intentamos entrar desde fuera sin iniciar sesión o registrarnos no nos va a dejar por uno de los dos motivos, para ello en settings en la aplicación principal debemos de crear una propiedad nueva debajo de STATIC\_URL que se va a llamar LOGIN\_URL  
y que le vamos a dar el valor de nuestra url de inicio de sesión para cuando alguien intente entrar desde fuera sin registrarse o iniciar sesión sea dirigido a esa url de inicio de sesión, donde también aparecerá la barra de navegación donde está el signup para registrarse por si aún no lo está.

STATIC\_URL = 'static/'

LOGIN\_URL = '/signin/'

Para pasar un proyecto a producción haremos lo siguiente:

* En settings nos vamos a secret\_key y le aplicamos el siguiente código que nos da la app web de Render.

*# SECURITY WARNING: keep the secret key used in production secret!*

SECRET\_KEY = os.environ.get('SECRET\_KEY', default='your secret key')

*#Esta secret\_key es la que se utiliza para las sesiones de los usuarios, por lo que es importante que sea segura y no se comparta con nadie.*

Después nos iremos a DEBUG y cambiaremos el TRUE por el siguiente código:

*# SECURITY WARNING: don't run with debug turned on in production!*

*#Cuando pasamos a producción este debug debe ser False, pero mientras estamos desarrollando la aplicación debe estar en True.*

DEBUG = 'RENDER' not in os.environ

*#El render es una variable de entorno que se utiliza para saber si estamos en producción o en desarrollo y cambiar el valor de DEBUG en consecuencia.*

Después:

ALLOWED\_HOSTS = []

RENDER\_EXTERNAL\_HOSTNAME = os.environ.get('RENDER\_EXTERNAL\_HOSTNAME')

if RENDER\_EXTERNAL\_HOSTNAME:

    ALLOWED\_HOSTS.append(RENDER\_EXTERNAL\_HOSTNAME)

*#Si la variable de entorno RENDER\_EXTERNAL\_HOSTNAME existe, entonces se añade a la lista de ALLOWED\_HOSTS. Esto es necesario para que la aplicación pueda ser accedida desde la URL que Render nos proporciona.*

Para cambiar a una base de datos por otro servidor, una base de datos de verdad por así decirlo, no sqlite3, debemos escribir en el terminal lo siguiente:

PS D:\Datos\CRUDenDjango> pip install dj-database-url psycopg2-binary

Después, siguiendo en settings, vamos a importar lo siguiente:

import dj\_database\_url

Después en lo que haya en DATABASES lo vamos a comentar porque lo vamos a cambiar.

DATABASES = {

    'default': {

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

*#'NAME': BASE\_DIR / 'db.sqlite3',*

    }

}

Siguiente paso ejecutar esto en el terminal:

PS D:\Datos\CRUDenDjango> pip install whitenoise[brotli]

Y después escribir en MIDDLEWARE hemos añadido el último código:

    'whitenoise.middleware.WhiteNoiseMiddleware',

MIDDLEWARE = [

    'django.middleware.security.SecurityMiddleware',

    'django.contrib.sessions.middleware.SessionMiddleware',

    'django.middleware.common.CommonMiddleware',

    'django.middleware.csrf.CsrfViewMiddleware',

    'django.contrib.auth.middleware.AuthenticationMiddleware',

    'django.contrib.messages.middleware.MessageMiddleware',

    'django.middleware.clickjacking.XFrameOptionsMiddleware',

    'whitenoise.middleware.WhiteNoiseMiddleware',

]

Después debajo de STATIC\_URL escribimos lo siguiente:

if not DEBUG:

*# Tell Django to copy static assets into a path called `staticfiles` (this is specific to Render)*

    STATIC\_ROOT = os.path.join(BASE\_DIR, 'staticfiles')

*# Enable the WhiteNoise storage backend, which compresses static files to reduce disk use*

*# and renames the files with unique names for each version to support long-term caching*

    STATICFILES\_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage'

Después creamos una carpeta build.sh fuera de todo y escribimos lo siguiente:

*#!/usr/bin/env bash*

*# Exit on error*

set -o errexit

*#poetry install*

python manage.py collectstatic --no-input

python manage.py migrate

y luego ejecutar el siguiente comando en el terminal de bash:

chmod a+x build.sh

después este otro:

pip install gunicorn uvicorn

después en el terminal de PowerShell:

pip freeze > requirements.txt

Después escribir en el build.sh:

pip install –r requirements.txt

Después creamos otro archivo llamado .gitignore y ahí meteremos los archivos que no queramos subir que serán los siguientes:

db.sqlite3

venv

\_\_pycache\_\_

Después “git init” y después “git add .” y después “git commit -m ‘first commit’”.

<button *class*="navbar-toggler" *type*="button" *data-bs-toggle*="collapse" *data-bs-target*="#navbarSupportedContent" *aria-controls*="navbarSupportedContent" *aria-expanded*="false" *aria-label*="Toggle navigation">

            <span *class*="navbar-toggler-icon"></span>

          </button>