UNIVERSIDAD NACIONAL DE COLOMBIA
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA DE SISTEMAS E INDUSTRIAL

Entrega 04

Fecha: 24/10/2025

Joan Camilo Betancourt Gonzalez (jobetancourtg@unal.edu.co)
Nicolás Alejandro Diosa Benavides (ndiosab@unal.edu.co)
Jonathan Felipe López Nuñez (jolopezn@unal.edu.co)
Raúl Felipe Rodríguez Hernández (rrodriguezhe@unal.edu.co)

---

1. TUTORIAL

Django desde 0

Contexto:

Django es un framework para desarrollo web en Python, es conocido por su facilidad y rapidez para el desarrollo con buenas prácticas de dichas aplicaciones.
Este framework tiene todas las herramientas que se podrían llegar a necesitar para una página web: ORM, sistema de autenticación, panel de administración, enrutamiento de URLs, plantillas html, seguridad integrada y servidor de desarrollo.
Para este tutorial se usará Docker para tener la base de datos de PostgreSQL en un contenedor, evidentemente Python como lenguaje principal, y Django como Framework junto con su ORM, Django ORM, además, se usará el editor de código VS-Code.

Creación base de datos:

Iniciaremos con la base de datos, y para eso, descargamos Docker, en windows podemos realizar este proceso con el siguiente comando en CMD:

```shell
winget install Docker.DockerDesktop
```

Aceptamos lo que nos pidan, esperamos y podemos confirmar la instalación con los siguientes comandos, si nos salen las versiones es que se instaló correctamente

Shell

```
docker --version
docker-compose --version
```

Un archivo .env se usa para guardar las variables de entorno para evitar exponer nuestras credenciales en el resto del código, otras partes del programa podrán acceder a nuestro archivo .env, pero no publicaremos este archivo, así cada programador tiene su propio archivo .env y no comparte sus credenciales.

None

```
# Configuración de PostgreSQL
POSTGRES_USER=usuario
POSTGRES_PASSWORD=contraseñasegura
POSTGRES_DB=nombreBaseDatos
POSTGRES_CONTAINER_NAME=nombreContenedor
POSTGRES_PORT=0811
```

Este es el formato que usaremos, en nuestro .env vamos a declarar características de nuestra base de datos, la idea es que cada desarrollador cree su propio archivo llamando ".env" y cambie las variables con datos diferentes.

Ahora vamos a crear un archivo con el nombre "docker-compose.yml", este archivo nos servirá para declarar como queremos que se cree nuestro contenedor de Docker.

```yaml
version: '3.8'

services:
  db:
    image: postgres:latest
    container_name: ${POSTGRES_CONTAINER_NAME}
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    ports:
      - "${POSTGRES_PORT}:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    restart: always

volumes:
  postgres_data:
```

Esta es una plantilla de docker-compose.yml, en resumen, lo que se hace es declarar que Docker necesita utilizar una plantilla de contenedor llamada postgres, y usará su versión más nueva, y con todos los campos con el formato ${...} se le está diciendo que saque esos datos

Ahora con los archivos .env y docker-compose.yml, y algún archivo .sql vamos a automatizar la creación de la base de datos con un archivo .bat, un archivo .bat lo que hace es ejecutar comandos en CMD de manera secuencial.

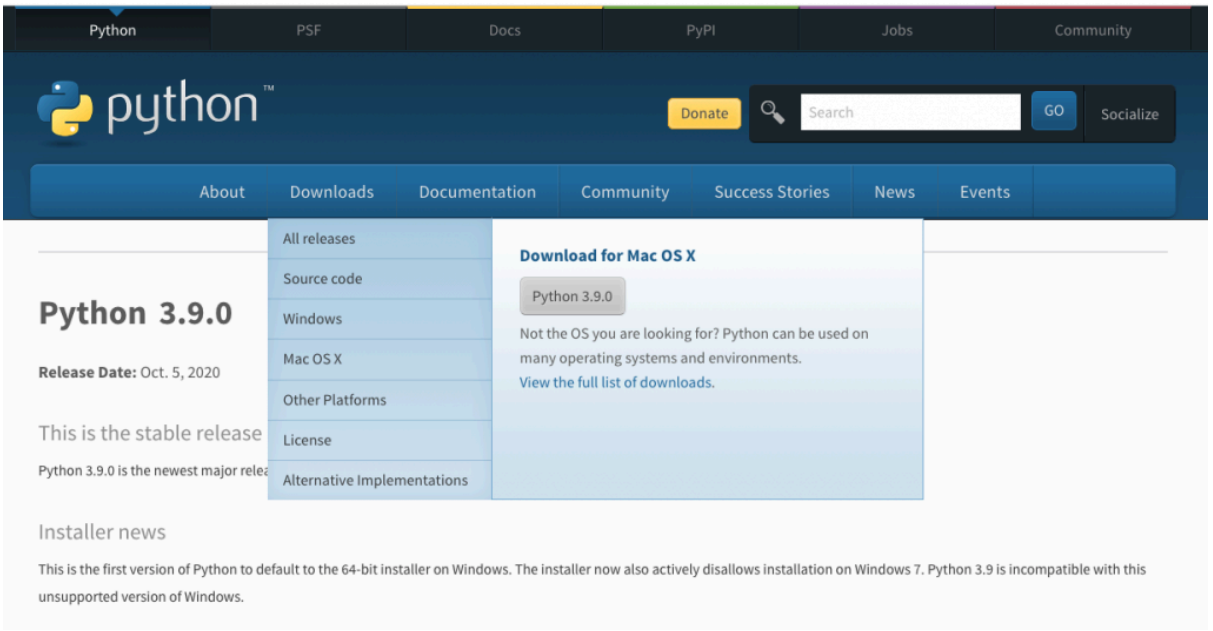
Este archivo setup.bat lo que hace es crear un contenedor con el comando docker compose up -d, este comando necesita tener docker desktop abierto, y lo que hace es crear un contenedor con las especificaciones dadas en docker-compose.yml, después de crear el contenedor, ejecuta el archivo Proyecto\Backend\cmd\bd\init.sql dentro del contenedor para inicializar la base de datos.

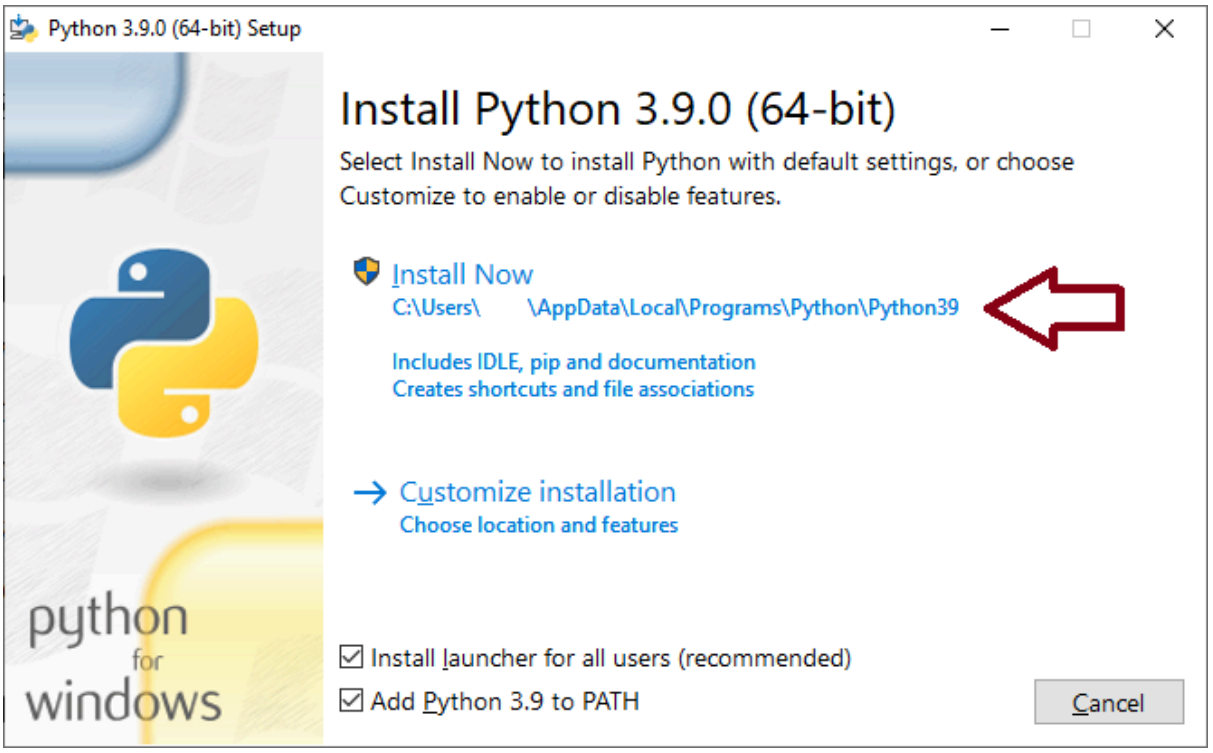
```
        @echo off
        :: Titulo de la ventana
        title Django Project Setup Script

        :: Print
        echo  Iniciando setup del proyecto...
        echo.

        :: Levantamiento de la Base de Datos con Docker Compose
        echo Levantando PostgreSQL con Docker Compose...
        docker compose up -d

        :: Si el anterior comando fallo, intentar con la version antigua
        IF %ERRORLEVEL% NEQ 0 (
            echo Intento fallido con 'docker compose', probando con 'docker-compose'...
            docker-compose up -d
        )
        :: Esperar
        echo Cargando...
        timeout /t 5 /nobreak > NUL

        :: Ejecutar el script SQL de inicialización
        :: Asegurarse de que estas variables coincidan con el .env
        set DB_USER=usuario
        set DB_NAME=parchate
        set CONTAINER_NAME=postgres_server

        IF EXIST "Proyecto\Backend\cmd\bd\init.sql" (
            echo Ejecutando script SQL de inicialización...
               type  "Proyecto\Backend\cmd\bd\init.sql"  |  docker  exec  -i  %CONTAINER_NAME%  psql  -U
%DB_USER% -d %DB_NAME%
        ) ELSE (
            echo No se encontró Proyecto\Backend\cmd\bd\init.sql
        )
        cd ..\..
        echo  Setup completado.
        :EOF
```

Instalación Python:

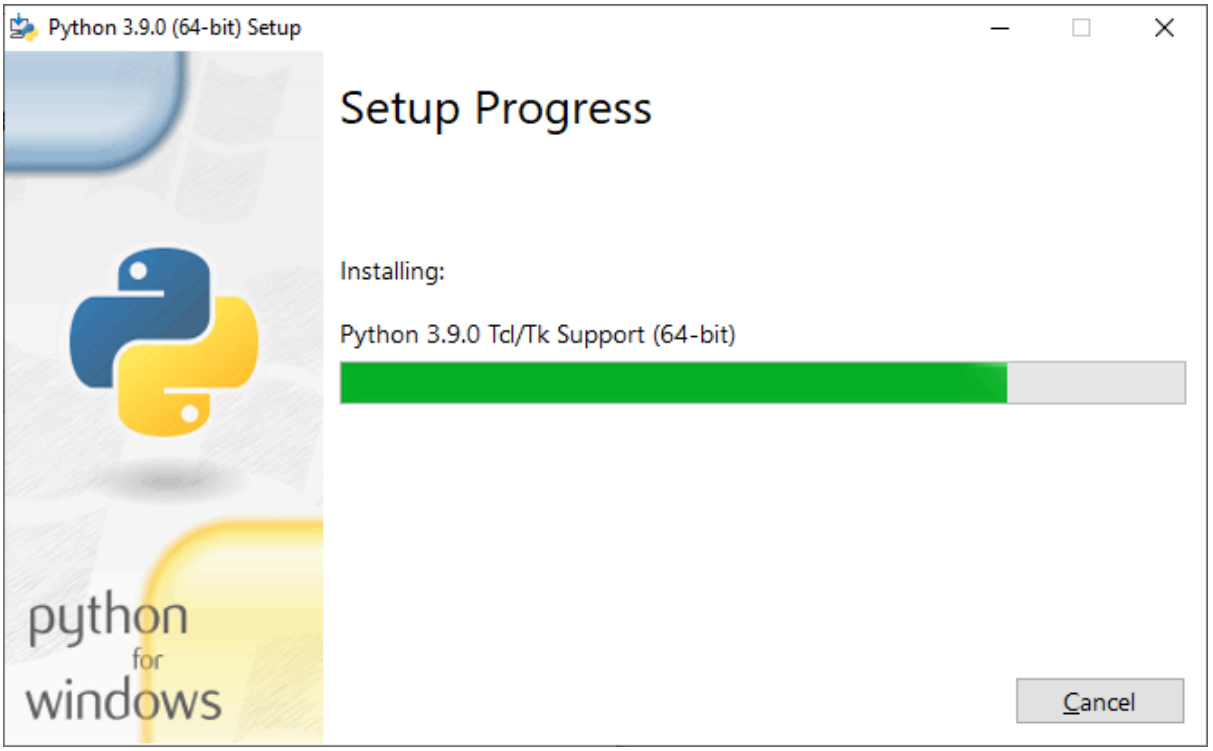
Ahora es necesario realizar la instalación de Python y Django.

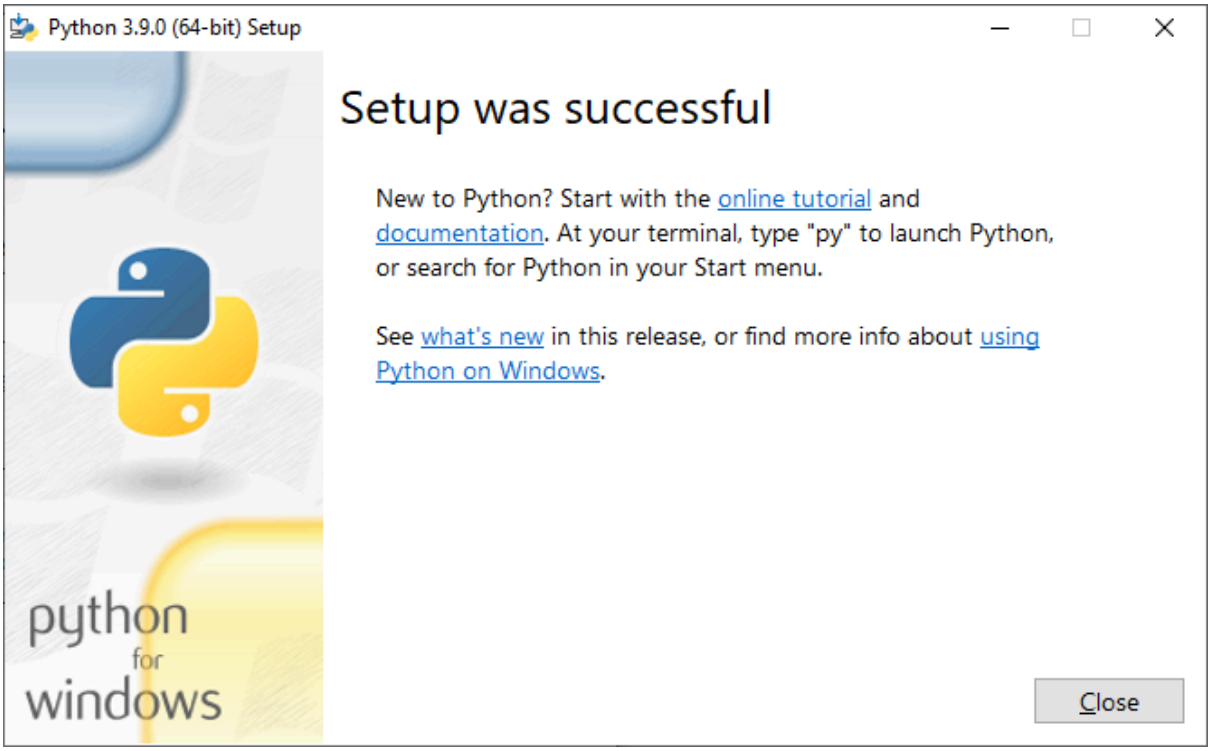
Para instalar Python es necesario ir al sitio de descargas oficial https://www.python.org/downloads/. Ya en el sitio se debe seleccionar el sistema operativo y la versión de Python deseadas





Ya en el instalador es muy importante seleccionar que Python se incluya en el path para facilitar la ejecución de módulos desde consola.





Para verificar la correcta instalación de Python se ejecuta el comando desde la cmd

```
pip --version
```

```
pip 25.2 from C:\Python313\Lib\site-packages\pip (python 3.13)
```

Entorno virtual:

Posteriormente a instalar python, se debe hacer la instalación de Django. Para esto, en la carpeta del proyecto a desarrollar se crea un entorno virtual para Python, cómo se desarrollará en VS Code, se ejecuta esta instrucción en la terminal donde está la carpeta:

```
None
python -m venv venv
```

Este comando crea un entorno virtual en la carpeta, de python, llamado venv, esto se hace por buenas prácticas, ya que permite preparar proyectos separadamente y gestionarlos de manera eficiente. Con esto en la carpeta del proyecto, se habrá generado una llamada venv que contiene las carpetas Include, Lib y Scripts. Ahora hay que activarlo con el siguiente comando:

```
None
venv\Scripts\activate
```

Se entra a la carpeta venv, Scripts y se ejecuta el archivo activate, haciendo esta instrucción debería aparecer en la terminal la línea de comandos de esta forma:

```
None
(venv)PS C:\...\...\...\...\...\...\nombre_proyecto>
```

Nota 1: En algunos casos no se puede activar de una vez, sino que toca hacer una configuración anterior. Si al intentar activar el entorno virtual aparece un error que dice "No se puede cargar el archivo ... porque la ejecución de scripts está deshabilitada en este sistema.", se debe hacer lo siguiente: abrir PowerShell como administrador , ejecutar el comando Set-ExecutionPolicy RemoteSigned -Scope CurrentUser y confirmar seleccionando la opción "Si". A continuación, se reinicia el VS Code y se activa el entorno.

Nota 2: Para realizar los siguientes comandos siempre se debe tener el entorno virtual activado, en caso de que se abra una nueva terminal, se requiere activar nuevamente el entorno.

Instalación Django:

Con el entorno virtual activado se instala Django con el siguiente comando:

None

```
pip install django psycopg2-binary python-dotenv
```

Se llama al instalador de paquetes de python y se procede a instalar django, un driver para conectar django con PostgreSQL y el python-dotenv para la utilización de las variables de entorno que se crearon en el .env.

Creación de proyecto y aplicación en Django:

Se procede con la creación de un proyecto en django con el siguiente comando:

None

```
django-admin startproject Internal
```

Esto creará una carpeta llamada Internal que contiene un archivo llamado manage.py y otro db.sqlite3. Si se ejecuta el archivo manage, dentro de Internal, con la instrucción python manage.py runserver, se abrirá una dirección ip que lleva a la página de inicio de nuestro proyecto en Django, sin embargo esto se realizará más adelante.

Nota 3: En caso de que su Internal quede dentro de alguna carpeta, será necesario tener una copia del archivo .env dentro de ésta carpeta para que funcione más adelante la migración de la base de datos.

Ahora, dentro de la carpeta Internal se ejecuta el comando:

None

```
python manage.py startapp core
```

Para crear la aplicación que se va a manipular, esto habrá generado una carpeta llamada core con diferentes carpetas y ejecutables, entre ellos apps, models y views, que se utilizaran más adelante, a continuación de crear la aplicación, se debe añadir al ejecutable settings.py que se encuentra en el proyecto creado, en este caso en internal, y se edita el código añadiendo el nombre de la aplicación en la lista de INSTALLED_APPS de la siguiente forma:

Python

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
```

```
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'core', #Aplicación añadida
]
```

Dentro de la carpeta internal se debe ejecutar el siguiente comando

None

```
 python manage.py runserver
```

Esto debe iniciar el servidor de desarrollo de Django, el cual se encarga de ejecutar la aplicación localmente para realizar pruebas y verificar su correcto funcionamiento.

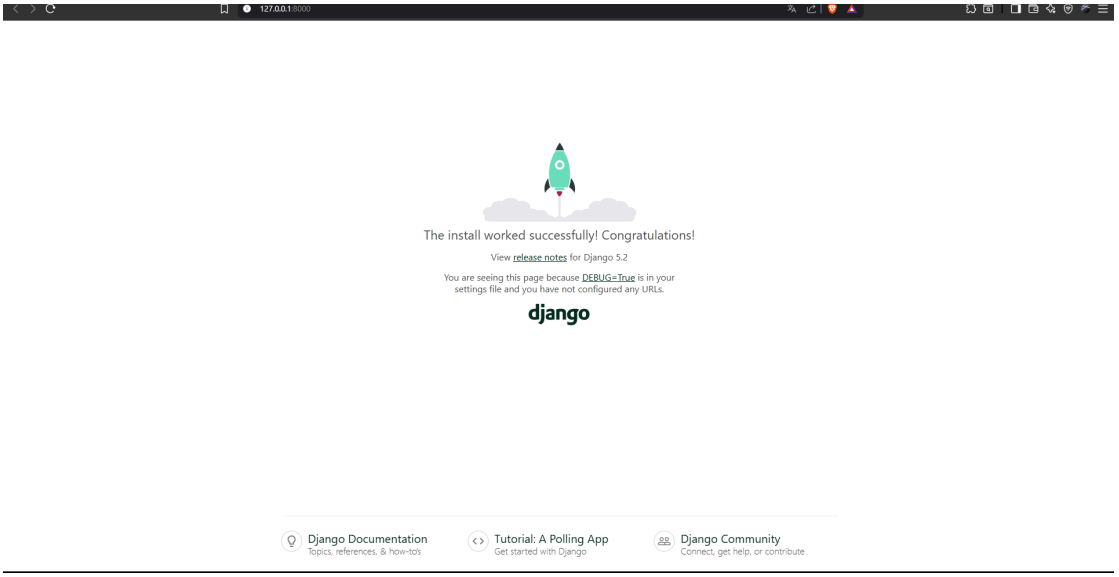
Se mostrará un mensaje como

None

```
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.
October 23, 2025 - 20:13:54
Django version 5.2.7, using settings 'Internal.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Lo importante de dicho mensaje es que la línea Starting development server at http://127.0.0.1:8000/ indica que el servidor está en ejecución y la aplicación Django puede visualizarse desde el navegador ingresando a la dirección http://127.0.0.1:8000/. Al acceder a

esa dirección, se mostrará la **página** de inicio predeterminada de Django, lo cual confirma que el entorno virtual, el framework y la estructura del proyecto están correctamente configurados.



Para detener el servidor es necesario oprimir Ctrl + C.

Conexión de Django con PostgreSQL:

Posteriormente se debe ubicar el archivo settings.py dentro de la carpeta principal del proyecto, que se encuentra en: Internal/Internal/settings.py.

En este punto, es necesario modificar la sección correspondiente a la base de datos para establecer la conexión con **PostgreSQL** que se ejecuta dentro del contenedor de **Docker**.
Para ello, se reemplaza el bloque predeterminado de configuración `DATABASES` por el siguiente código:

Python

```python
import os
from pathlib import Path
from dotenv import load_dotenv

BASE_DIR = Path(__file__).resolve().parent.parent
load_dotenv(BASE_DIR / '.env')
```

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.getenv('POSTGRES_DB'),
        'USER': os.getenv('POSTGRES_USER'),
        'PASSWORD': os.getenv('POSTGRES_PASSWORD'),
        'HOST': os.getenv('POSTGRES_HOST', 'localhost'),
        'PORT': os.getenv('POSTGRES_PORT', '5432'),
    }
}
```

Con esta configuración, Django utiliza las variables definidas en el archivo .env ubicado en la raíz del proyecto, donde se especifican las credenciales y el puerto del contenedor PostgreSQL. El uso de python-dotenv permite mantener estos valores separados del código fuente, lo cual mejora la seguridad y facilita el despliegue en diferentes entornos.

Ya con esto Django sabe a que base de datos conectarse, entonces se ejecuta

None

```
python manage.py migrate
```

Esto ordena a Django aplicar todas las migraciones iniciales, que son los scripts encargados de crear las tablas necesarias dentro de la base de datos PostgreSQL configurada.

Durante este proceso, Django establece la conexión con el contenedor de Docker y crea automáticamente las estructuras de datos correspondientes a los módulos base del framework, tales como:

auth: manejo de usuarios y autenticación

admin: interfaz de administración

contenttypes: tipos de contenido dinámico

sessions: gestión de sesiones de usuario

Si la conexión a PostgreSQL es exitosa, en la terminal se mostrará un mensaje similar a:

None

```
Applying admin.0001_initial... OK
Applying auth.0001_initial... OK
Applying contenttypes.0001_initial... OK
Applying sessions.0001_initial... OK
```

Esto indica que todas las migraciones se ejecutaron correctamente y que Django ya está trabajando con la base de datos alojada dentro del contenedor Docker.

Para verificar desde Docker que la base de datos tenga tablas se ejecuta en una nueva terminal fuera de venv:

None

```
docker exec -it postgres_server psql -U usuario -d parchate
```

Donde postgres_server corresponde al nombre del contenedor definido en el archivo docker-compose.yml. usuario es el nombre de usuario configurado en el archivo .env y parchate es el nombre de la base de datos que se está utilizando en el proyecto.

Luego se ejecuta el comando `\dt`. Este comando lista todas las tablas existentes dentro del esquema público de la base de datos.

Si la conexión y las migraciones se realizaron correctamente, se visualizará una salida similar a la siguiente:

```
parchate=# \d
                    List of relations
 Schema |              Name                  |   Type   |  Owner
--------+------------------------------------+----------+---------
 public | actividades                        | table    | usuario
 public | auth_group                         | table    | usuario
 public | auth_group_id_seq                  | sequence | usuario
 public | auth_group_permissions             | table    | usuario
 public | auth_group_permissions_id_seq      | sequence | usuario
 public | auth_permission                    | table    | usuario
 public | auth_permission_id_seq             | sequence | usuario
 public | auth_user                          | table    | usuario
 public | auth_user_groups                   | table    | usuario
 public | auth_user_groups_id_seq            | sequence | usuario
 public | auth_user_id_seq                   | sequence | usuario
 public | auth_user_user_permissions         | table    | usuario
 public | auth_user_user_permissions_id_seq  | sequence | usuario
 public | chats                              | table    | usuario
 public | django_admin_log                   | table    | usuario
 public | django_admin_log_id_seq            | sequence | usuario
 public | django_content_type                | table    | usuario
 public | django_content_type_id_seq         | sequence | usuario
 public | django_migrations                  | table    | usuario
 public | django_migrations_id_seq           | sequence | usuario
 public | django_session                     | table    | usuario
 public | eventos_calendario                 | table    | usuario
 public | materias                           | table    | usuario
 public | participantes_actividad            | table    | usuario
 public | tareas                             | table    | usuario
 public | usuarios                           | table    | usuario
(26 rows)
```

Modelos en Django:

Una vez verificado que la base de datos está correctamente creada en docker y que django tambien la esta reconociendo, podemos empezar a generar los modelos que son la representación de las tablas de la base de datos en clases para su uso en el ORM y el CRUD de django. Para esto, en la carpeta Core del proyecto se debe ubicar el archivo models.py y alli empezar a crear las clases para cada tabla, por ejemplo:

SQL

```sql
CREATE TABLE "usuarios" (
  "id" uuid PRIMARY KEY DEFAULT (gen_random_uuid()),
  "nombre_usuario" varchar(30) UNIQUE NOT NULL,
  "email" varchar(255) UNIQUE NOT NULL,
  "contrasena" varchar(255) NOT NULL,
  "nombre" varchar(100),
  "bio" varchar(280),
  "foto_perfil" text
);
```

Esto es lo que se tiene en la base de datos de Parchate, más específicamente dentro del archivo init.sql, es una creación de tabla llamada usuarios con columnas: id, nombre_usuario, email, contrasena, nombre, bio, foto_perfil. Ahora, el modelo de esta tabla en models.py se veria asi:

Python

```
class Usuario(models.Model):
   id=models.UUIDField(primary_key=True, default=uuid.uuid4,
editable=False)
     nombre_usuario    =    models.CharField(max_length=30,
unique=True)
    email = models.CharField(max_length=255, unique=True)
    contrasena = models.CharField(max_length=255)
      nombre  =  models.CharField(max_length=100,  blank=True,
null=True)
       bio  =  models.CharField(max_length=280,  blank=True,
null=True)
    foto_perfil = models.TextField(blank=True, null=True)

    class Meta:
        managed = False
        db_table = 'usuarios'
```

Con esto se está creando la clase Usuario que seria la equivalencia a la tabla anteriormente mencionada, como se ve hereda de models el modelo correspondiente de la base de datos, y para cada columna se crea una variable equivalente mientras se indica el tipo de campo de cada una mediante los métodos de models. Se puede configurar si se quiere que sea llave primaria, la longitud de entrada, un predeterminado para los valores, si puede estar vacío, entre otras características. Finalmente, a cada modelo se le asigna class Meta, ya que estas tablas fueron creadas por alguien externo y traídas de otro lugar ajeno a django, en este caso postgreSQL, esta clase permite configurar si como usuarios queremos que django tenga el control de configurarlas por nosotros, sin embargo lo único que necesitamos de django es la posibilidad de hacer CRUD, no de editar las tablas en sí. Así, se van creando los modelos para cada tabla.

Instanciación y prueba de datos en PostgreSQL:

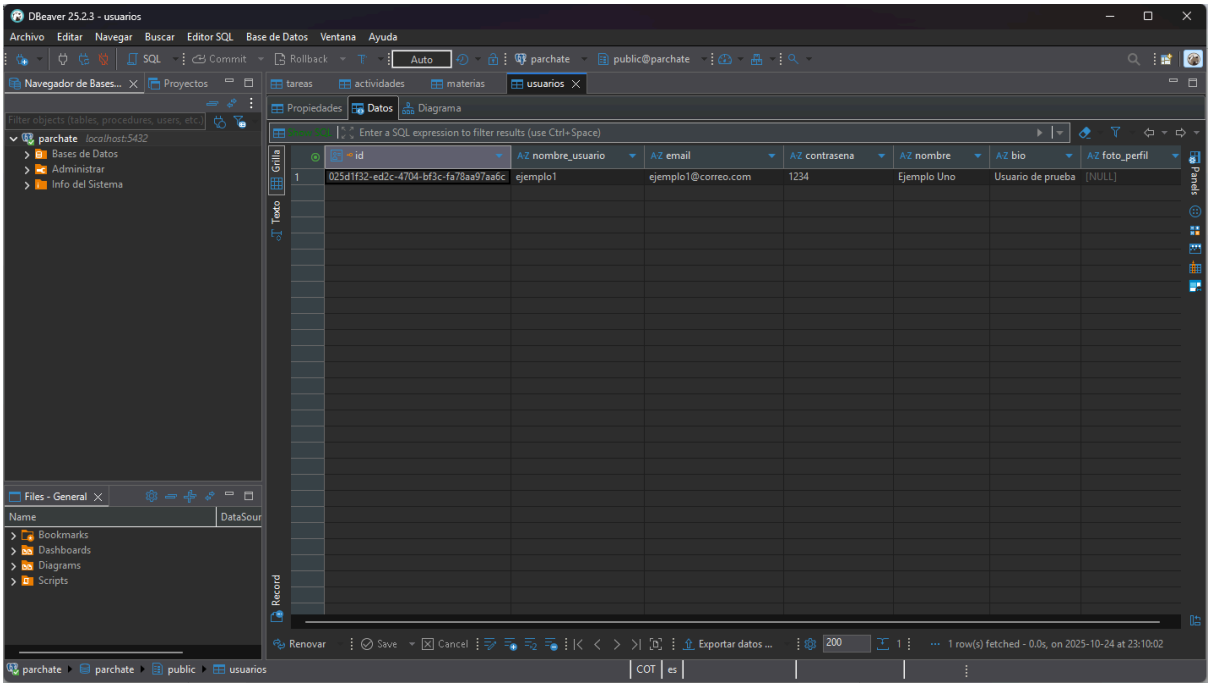
A través del siguiente código probamos las clases de models, y nuestra instancia DB_manager, que es parte de la implementación del patrón de diseño Singleton y funcionara

para hacer el CRUD necesario de los datos de django a postgreSQL, viendo los resultados a través de la interfaz gráfica de DBeaver conectada a nuestra base de datos de Docker:

Python

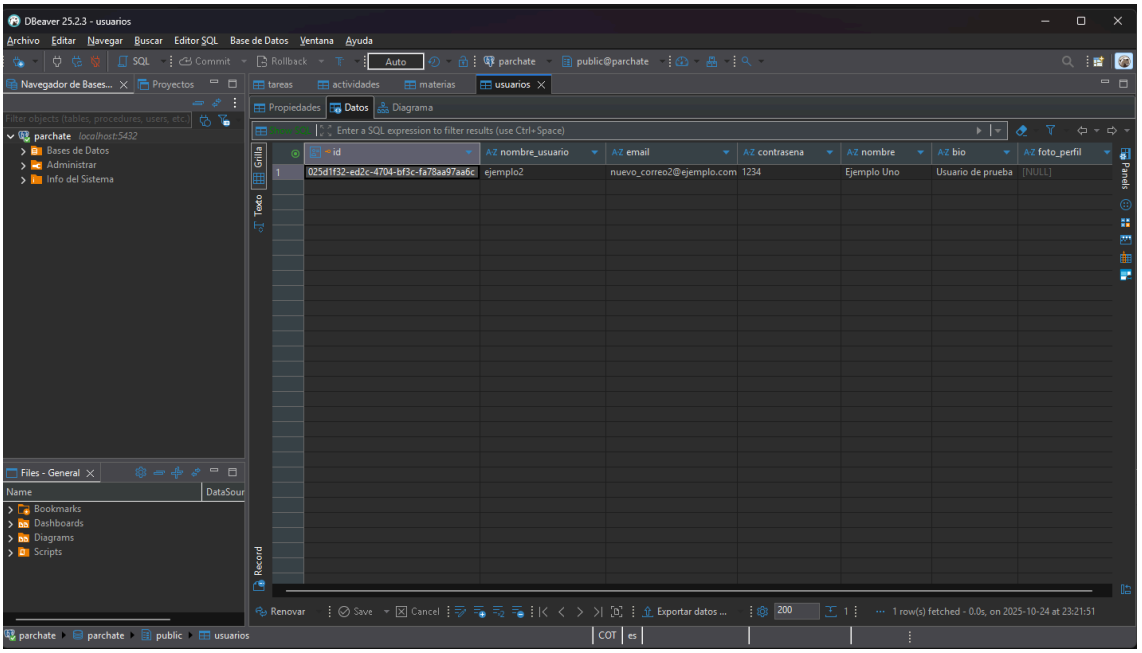
```
from core.Persistencia.DB_manager import DB_Manager

db = DB_Manager()

#Create usuario 1ERA EJECUCIÓN
db.create_usuario(nombre_usuario="ejemplo1",email="ejemplo1@
correo.com",contrasena="1234",nombre="Ejemplo
Uno",bio="Usuario de prueba",foto_perfil=None)

#Read 2DA EJECUCIÓN

db.get_usuario_by_nombre_usuario("ejemplo1")

#Update usuario existente 3ERA EJECUCIÓN
db.update(Usuario,       "nombre_usuario",       "ejemplo1",
nombre_usuario                =                "ejemplo2",
email="nuevo_correo2@ejemplo.com")

#Borrar usuario creado 4TA EJECUCIÓN
db.delete(Usuario, "nombre_usuario", "ejemplo2")
```

El resultado de la primera ejecución en el DBeaver es el siguiente:



El de la segunda ejecución en el shell de Django es:

El de la tercera ejecución en el DBeaver es:



El de la cuarta ejecución en el DBeaver es: