EXERCISES QUE TRABAJAREMOS EN EL CUE

- **EXERCISE 1: CONEXIÓN DE DJANGO A UNA BASE DE DATOS POSTGRESQL**
- **EXERCISE 2: CREANDO LA BASE DE DATOS Y EL USUARIO**
- **EXERCISE 3: INSTALANDO DJANGO EN UN AMBIENTE VIRTUAL**
- **EXERCISE 4: MIGRACIÓN DE LA BASE DE DATOS Y PROBANDO EL PROYECTO**
- **EXERCISE 5: CREANDO MODELOS EN DJANGO**
- **EXERCISE 6: ACTUALIZANDO EL SETTINGS.PY Y EJECUTANDO LAS MIGRACIONES**
- **EXERCISE 7: VERIFICAMOS EL ESQUEMA EN LA BASES DE DATOS POSTGRESQL**

EXERCISE 1: CONEXIÓN DE DJANGO A UNA BASE DE DATOS POSTGRESQL

Para esta práctica debemos tener instalado el virtualenvwrapper, donde procederemos a instalar las dependencias para nuestro proyecto de ORM con PostgreSQL. Seguidamente, crearemos un nuevo entorno virtual con el comando `mkvirtualenv`.

INSTALACIÓN DE POSTGRESQL EN LINUX UBUNTU:

Primero instalará los componentes esenciales. Esto incluye pip, el administrador de paquetes de Python para instalar y administrar componentes de Python, y también el software de la base de datos con sus bibliotecas asociadas.

Trabajaremos con Python 3, que viene con Ubuntu 20.04. Se inicia su instalación escribiendo:

```
$ sudo apt update

$ sudo apt install python3-pip python3-dev libpq-dev postgresql
postgresql-contrib
```

EXERCISE 2: CREANDO LA BASE DE DATOS Y EL USUARIO

De forma predeterminada, Postgres utiliza un esquema de autenticación llamado "autenticación de pares" para las conexiones locales. Básicamente, esto significa que si el nombre de usuario del sistema operativo coincide con un nombre de usuario de Postgres válido, ese usuario puede iniciar sesión sin más autenticación.

Durante la instalación de Postgres, se creó un usuario del sistema operativo llamado `postgres` para corresponder al usuario administrativo de PostgreSQL. Debe utilizar este usuario para realizar tareas administrativas. Puede usar `sudo`, y pasar el nombre de usuario con la opción `-u`.

Inicie sesión en una sesión interactiva de Postgres escribiendo:

```
$ sudo -u postgres psql
```

Primero, creará una base de datos para el proyecto Django. Cada proyecto debe tener su propia base de datos aislada por razones de seguridad. Para este ejemplo, llamaremos a la base de datos **project-orm-django**:

```
postgres=# CREATE DATABASE project_orm_django;
```

Seguidamente, se creará un usuario de base de datos que se utilizará para conectarse e interactuar con la base de datos:

```
postgres=# CREATE USER userdjango WITH PASSWORD 'userdjango';
```

Luego, se modificarán algunos de los parámetros de conexión para el usuario que acaba de crear. Esto acelerará las operaciones de la base de datos para que no sea necesario consultar y establecer los valores correctos cada vez que se establece una conexión.

```
postgres=# ALTER ROLE userdjango SET client_encoding TO 'utf8';
postgres=# ALTER ROLE userdjango SET default_transaction_isolation TO
'read committed';
postgres=# ALTER ROLE userdjango SET timezone TO 'UTC';
```

Se ha configurado la codificación predeterminada en UTF-8, que espera Django. También está configurado el esquema de aislamiento de transacciones predeterminado en "lectura confirmada", que bloquea las lecturas de transacciones no confirmadas. Por último, se está configurando la zona horaria. De manera predeterminada, sus proyectos de Django se configurarán para usar UTC. Todas estas son recomendaciones del propio proyecto Django.

Se procede a otorgar los permisos de acceso de usuario a la base de datos que se creó:

```
postgres=# GRANT ALL PRIVILEGES ON DATABASE project_orm_django TO
userdjango;
```

Salimos de la terminal de postgresql:

```
postgres=# \q
```

EXERCISE 3: INSTALANDO DJANGO EN UN AMBIENTE VIRTUAL

Una vez que hayas instalado virtualenvwrapper, se procede a crear un nuevo entorno virtual con el comando `mkvirtualenv`.

```
$ mkvirtualenv django-orm-postgresql
```

Al finalizar observamos que el nuevo entorno virtual se encuentra activo (`django-orm-postgresql`), y verificamos qué paquetes tiene instalado el previamente creado:

```
$ pip list
Package    Version
---------- -------
pip        22.2.2
setuptools 64.0.3
wheel      0.37.1
```

Procedemos a instalar tanto Django, como el conector a postgresql, el driver `psycopg2`:

```
$ pip install django
$ pip install psycopg2
```

Listamos los paquetes instalados:

```
$ pip list
Package            Version
------------------ -------
asgiref            3.5.2
backports.zoneinfo 0.2.1
Django             4.1.1
pip                22.2.2
psycopg2           2.9.3
setuptools         64.0.3
sqlparse           0.4.2
style              1.1.0
update             0.0.1
wheel              0.37.1
```

Procedemos a crear un proyecto en Django dentro del directorio `django-orm-postgresql`. Creando dentro del mismo un directorio `config`, se generará un script de administración en el directorio actual.

Creamos la carpeta del proyecto:

```
$ mkdir django-orm-postgresql
```

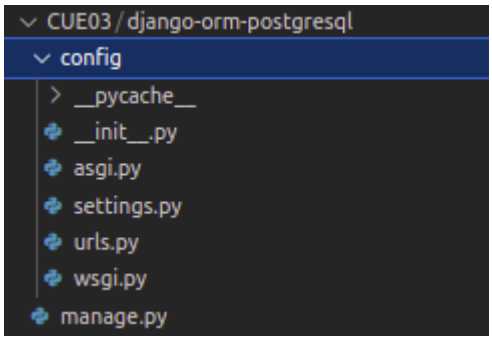
Creamos el proyecto:

```
django-admin startproject config .
```

Abrimos Visual Studio Code (VSC):

```
code .
```

Tenemos la siguiente estructura:



Ajustamos la bases de datos para conectarnos de Django a PostgreSQL. Al ya tener el proyecto creado, debemos configurarlo para usar la base de datos que creó.

Abrimos el archivo principal de configuración del proyecto Django llamado `settings.py`, que está ubicado dentro del directorio config del proyecto, y buscamos la sección Database:

CONFIG/SETTINGS.PY

```
# Database
# https://docs.djangoproject.com/en/4.1/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
```

```
    }
}
```

Actualmente está configurado para usar SQLite como base de datos en el proyecto. Se debe cambiar esto para que se use PostgreSQL como base de datos.

Para ello se cambia el motor para que use el adaptador `postgresql`, en lugar del adaptador `sqlite3`. Para la variable `NAME`, use el nombre de su base de datos (`project_orm_django` en esta práctica). También debe agregar las credenciales de inicio de sesión. Necesita el nombre de usuario, la contraseña, y el host para conectarse. Agregarás y dejarás en blanco la opción de puerto para que se seleccione el predeterminado.

CONFIG/SETTINGS.PY

```
# Database
# https://docs.djangoproject.com/en/4.1/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'project_orm_django',
        'USER': 'userdjango',
        'PASSWORD': 'userdjango',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

EXERCISE 4: MIGRACIÓN DE LA BASE DE DATOS Y PROBANDO EL PROYECTO

Ahora que Django está configurado, puede migrar sus estructuras de datos a su base de datos, y probar el servidor. Puede comenzar creando y aplicando migraciones a su base de datos. Como aún no tiene datos reales, esto simplemente configura la estructura inicial:

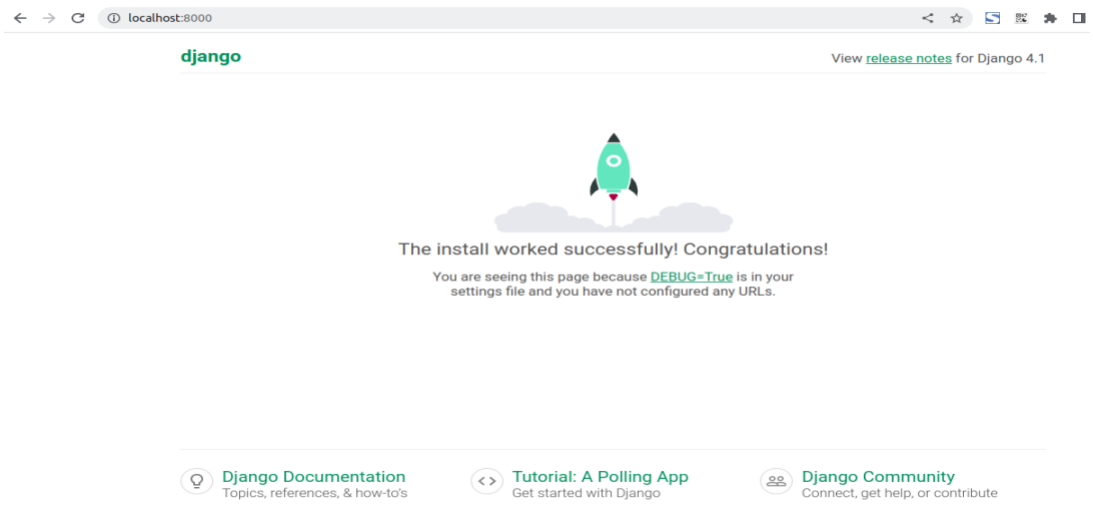
```
(django-orm-postgresql)$ python manage.py makemigrations
(django-orm-postgresql)$ python manage.py migrate
```

Después de crear la estructura de la base de datos, puede crear una cuenta administrativa escribiendo:

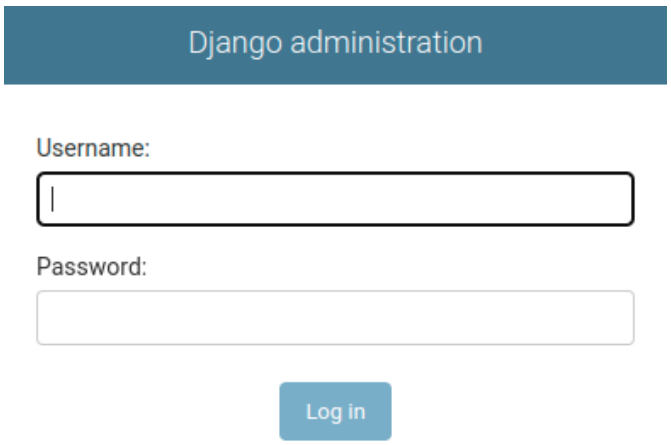
```
(django-orm-postgresql)$ python manage.py createsuperuser
```

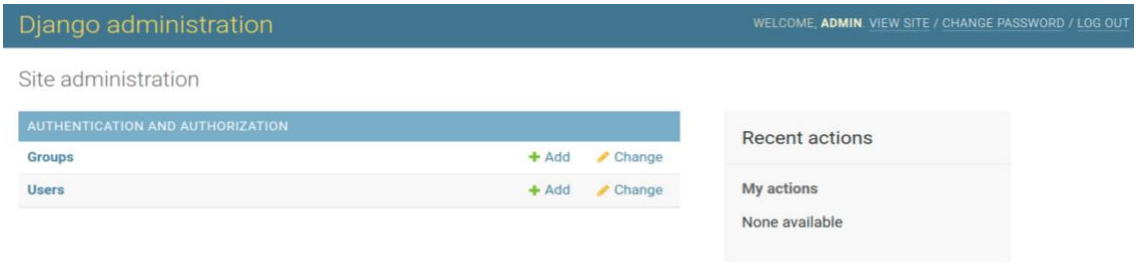
Luego, ejecutamos el proyecto en su servidor Web:

```
(django-orm-postgresql)$ python manage.py runserver
```



Ingresamos al sitio administrativo de Django (http://localhost:8000/admin/) con las credenciales previamente creadas del superuser:



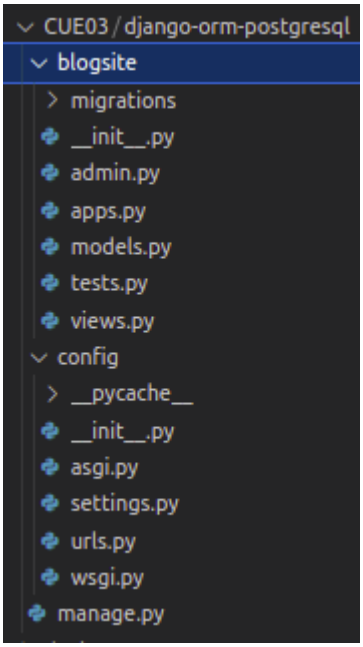


EXERCISE 5: CREANDO MODELOS EN DJANGO

Procederemos a crear una aplicación para nuestro proyecto, una aplicación de blog.

```
(django-orm-postgresql)$ python manage.py startapp blogsite
```

Tendremos la siguiente estructura del proyecto:



En esta parte de la práctica nos centraremos en el archivo `models.py`, que se encuentra en el directorio del sitio del blog (blogsite).

Primero necesitamos abrir y editar el archivo `models.py`, para que contenga el código que permite generar un modelo de un Post. Un modelo Post contiene los siguientes campos de bases de datos:

- `title:` el título de la entrada del blog.
- `slug:` donde se almacenan y generan direcciones URL válidas para páginas web.
- `content:` el contenido textual de la publicación del blog.
- `created_on:` la fecha en la que se creó la publicación.
- `author:` la persona que ha escrito la publicación.

Procedemos a crear el modelo:

```
from django.db import models
from django.template.defaultfilters import slugify
from django.contrib.auth.models import User
from django.urls import reverse


class Post(models.Model):
    title = models.CharField(max_length=255)
    slug = models.SlugField(unique=True, max_length=255)
    content = models.TextField()
    created_on = models.DateTimeField(auto_now_add=True)
    author = models.TextField()

    def get_absolute_url(self):
        return reverse('blog_post_detail', args=[self.slug])

    def save(self, *args, **kwargs):
        if not self.slug:
            self.slug = slugify(self.title)
        super(Post, self).save(*args, **kwargs)

    class Meta:
        ordering = ['created_on']

        def __unicode__(self):
            return self.title
```

Dentro de este archivo, se agregó el código para importar la API de modelos. Luego, importamos `slugify` para generar slugs a partir de cadenas, el usuario de Django para la autenticación, y viceversa, desde django.urls para brindarnos una mayor flexibilidad con la creación de URL.

Luego, agregamos el método de clase en la clase modelo que llamaremos Post, con los siguientes campos de bases de datos: `title`, `slug`, `content`, `created_on`, y `author`. A continuación,

agregaremos la funcionalidad para la generación de la URL, y la función para guardar el post. Esto es crucial, pues crea un enlace único para que coincida con nuestro post único.

Ahora, debemos indicarle al modelo cómo deben ordenarse las publicaciones y mostrarse en la página web. La lógica para esto se agregará a una clase `Meta` interna anidada, la cual generalmente contiene otra lógica de modelo importante que no está relacionada con la definición de campo de la base de datos.

EXERCISE 6: ACTUALIZANDO EL SETTINGS.PY Y EJECUTANDO LAS MIGRACIONES

Ahora que hemos agregado modelos a nuestra aplicación, debemos informar al proyecto de la existencia de la aplicación del sitio de blogs que acabamos de agregar, a la sección `INSTALLED_APPS` en `settings.py`.

```
# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blogsite',
]
```

Luego, ejecute el comando makemigrations en `manage.py`.

```
$ python manage.py makemigrations
Migrations for 'blogsite':
  blogsite/migrations/0001_initial.py
    - Create model Post
```

Como hemos creado un archivo de migración, debemos aplicar los cambios que éstos describen a la base de datos usando el comando `migrate`. Pero primero verifiquemos qué migraciones existen actualmente, usando el comando `showmigrations`.

```
$ python manage.py showmigrations
admin
 [X] 0001_initial
 [X] 0002_logentry_remove_auto_add
```

```
 [X] 0003_logentry_add_action_flag_choices
auth
 [X] 0001_initial
 [X] 0002_alter_permission_name_max_length
 [X] 0003_alter_user_email_max_length
 [X] 0004_alter_user_username_opts
 [X] 0005_alter_user_last_login_null
 [X] 0006_require_contenttypes_0002
 [X] 0007_alter_validators_add_error_messages
 [X] 0008_alter_user_username_max_length
 [X] 0009_alter_user_last_name_max_length
 [X] 0010_alter_group_name_max_length
 [X] 0011_update_proxy_permissions
 [X] 0012_alter_user_first_name_max_length
blogsite
 [ ] 0001_initial
contenttypes
 [X] 0001_initial
 [X] 0002_remove_content_type_name
sessions
 [X] 0001_initial
```

Notará que todas las migraciones están marcadas excepto la de **`0001_initial`**, que acabamos de crear con los modelos Post.

Ahora veamos qué sentencias SQL se ejecutarán una vez que hagamos las migraciones, usando el siguiente comando, que toma como argumento la migración y su título:

```
$ python manage.py sqlmigrate blogsite 0001_initial
BEGIN;
--
-- Create model Post
--
CREATE TABLE "blogsite_post" ("id" bigint NOT NULL PRIMARY KEY
GENERATED BY DEFAULT AS IDENTITY, "title" varchar(255) NOT NULL, "slug"
varchar(255) NOT NULL UNIQUE, "content" text NOT NULL, "created_on"
timestamp with time zone NOT NULL, "author" text NOT NULL);
CREATE INDEX "blogsite_post_slug_ad1b9573_like" ON "blogsite_post"
("slug" varchar_pattern_ops);
COMMIT;
```

Realicemos las migraciones para que se apliquen a nuestra base de datos PostgreSQL.

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, blogsite, contenttypes, sessions
Running migrations:
```

```
  Applying blogsite.0001_initial... OK
```

EXERCISE 7: VERIFICAMOS EL ESQUEMA EN LA BASES DE DATOS POSTGRESQL

Con las migraciones completas, deberíamos verificar la generación exitosa de las tablas PostgreSQL que hemos creado a través de nuestro modelo Django Post.

Para hacerlo, ejecute el siguiente comando en la terminal para iniciar sesión en PostgreSQL.

```
$ python manage.py migrate
$ psql -h postgres -U root

Password for user root:
psql (14.2 (Debian 14.2-1.pgdg110+1))
Type "help" for help.

root=#
```

Listamos la base de datos:

```
root=# \l
                                 List of databases
        Name         | Owner | Encoding |   Collate   |    Ctype    |  Access privileges
---------------------+-------+----------+-------------+-------------+---------------------
 postgres            | root  | UTF8     | en_US.utf8 | en_US.utf8 |
 project_orm_django  | root  | UTF8     | en_US.utf8 | en_US.utf8 | =Tc/root           +
                     |       |          |            |            | root=CTc/root      +
                     |       |          |            |            | userdjango=CTc/root
 root                | root  | UTF8     | en_US.utf8 | en_US.utf8 |
 template0           | root  | UTF8     | en_US.utf8 | en_US.utf8 | =c/root            +
                     |       |          |            |            | root=CTc/root
 template1           | root  | UTF8     | en_US.utf8 | en_US.utf8 | =c/root            +
                     |       |          |            |            | root=CTc/root
(5 rows)

root=#
```

Nos conectamos a la base de datos **`project_orm_django`**:

```
root=# \c project_orm_django
You are now connected to database "project_orm_django" as user "root".
project_orm_django=#
```

Listamos las tablas que se encuentran en la base de datos:

```
project_orm_django=# \dt
                    List of relations
 Schema |             Name              | Type  |    Owner
--------+-------------------------------+-------+-------------
 public | auth_group                    | table | userdjango
 public | auth_group_permissions        | table | userdjango
 public | auth_permission               | table | userdjango
 public | auth_user                     | table | userdjango
 public | auth_user_groups              | table | userdjango
 public | auth_user_user_permissions    | table | userdjango
 public | blogsite_post                 | table | userdjango
 public | django_admin_log              | table | userdjango
 public | django_content_type           | table | userdjango
 public | django_migrations             | table | userdjango
 public | django_session                | table | userdjango
(11 rows)
```

Verificamos los atributos que contiene la tabla **`blogsite_post`**:

```
project_orm_django=# \d blogsite_post
                                Table "public.blogsite_post"
   Column   |           Type           | Collation | Nullable |             Default
------------+--------------------------+-----------+----------+---------------------------------
 id         | bigint                   |           | not null | generated by default as identity
 title      | character varying(255)   |           | not null |
 slug       | character varying(255)   |           | not null |
 content    | text                     |           | not null |
 created on | timestamp with time zone |           | not null |
 author     | text                     |           | not null |
Indexes:
    "blogsite_post_pkey" PRIMARY KEY, btree (id)
    "blogsite_post_slug_ad1b9573_like" btree (slug varchar_pattern_ops)
    "blogsite_post_slug_key" UNIQUE CONSTRAINT, btree (slug)
```

Hemos verificado que las tablas de la base de datos se generaron con éxito a partir de nuestras migraciones de modelos de Django.