Tutorial para la construcción de aplicación Web usado Django , Python y docker.

En este tutorial vamos a ver cómo se realiza la construcción de una aplicación web usando Django , el levantamiento y conexión a la base de datos en PostgreSQL, configuración y levantamiento de Docker , y un "Hola mundo" que muestre una interfaz visual y se realice una ejecución que utilice todos los aspectos mencionados anteriormente.

Conocimientos necesarios para realizar el tutorial:

- Sintaxis de python intermedio , uso de clases y funciones
- Sintaxis de SQL : Creación de archivos .sql , conocimiento de sintaxis de QUERYS

Lenguaje: Python

Framework: Django

ORM: Django ORM (integrado por defecto, basado en clases -> tablas)

A. Configuración y levantamiento de la base de datos:

1. Para la configuración y levantamiento de la base de datos tenemos que tener el archivo **init.sql** con la creación de la tabla de datos, la creación de este archivo no se explicará ya que se espera el usuario ya sepa crearla. Este es un ejemplo de cómo debe verse el archivo:

```sql
SQL
CREATE TABLE "assignment" (
  "id" int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  "type" material_type,
  "max_score" int,
  "due_date" timestamptz
);

CREATE TABLE "material" (
  "id" int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
  "type" material_type,
  "file" bytea
);
```

2. Ahora vamos a crear un archivo **Dockerfile** en el cual, vamos a ejecutar el siguiente comando:

```
RUN apt-get update && apt-get install -y postgresql-client && rm -rf /var/lib/apt/lists/*
```

Este código Instala PostgreSQL client el cual es necesario para almacenar el archivo en el docker.

B. Contenido de los archivos .yml necesarios junto con archivos .env y requirements.txt

1. Hay un único archivo .yml el cual nombramos **docker-compose.yml** en el cual definimos 2 servicios, empezamos poniendo services:

Web:

```yaml
web:
  build: .
  container_name: django_web
  working_dir: /app
  command: sh -c "python manage.py runserver 0.0.0.0:8000 || tail -f /dev/null"
  ports:
    - "8000:8000"
  volumes:
    - ./Proyecto/SAFE:/app
  environment:
    - PYTHONUNBUFFERED=1
  depends_on:
    db:
      condition: service_healthy
  env_file:
    - .env
```

En este código las variables a configurar son las de **container_name** , en **command** la parte de runserver 0.0.0.0 : #puerto que vamos a usar# , **ports** #puerto a usar : el mismo repetido#

El resto de variables se dejan por default como aparecen en la foto.

DB:

```yaml
db:
  image: postgres:15
  container_name: postgres_db
  restart: unless-stopped
  environment:
    POSTGRES_USER: ${DB_USER}
    POSTGRES_PASSWORD: ${DB_PASSWORD}
    POSTGRES_DB: ${DB_NAME}
  ports:
    - "${DB_PORT}:5432"
  volumes:
    - pg_data:/var/lib/postgresql/data
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U ${DB_USER}"]
    interval: 5s
    timeout: 5s
    retries: 5
  env_file:
    - .env
```

En esté código imagen representa la imagen de la base de datos que vamos a usar, en este caso usaremos postgres:15 lo cual indica la versión, también podemos setear el nombre del contenedor y su environment: donde seriamos el usuario, contraseña y nombre de la base de datos, en ports se configura el puerto a usar por la base de datos, el resto de parámetros se dejan con esos valores por defecto.

De momento no se visualiza muy bien el porqué de cada archivo pero a medida que vayamos avanzando en el tutorial irá teniendo más claridad y sentido.

2. Vamos a crear un archivo requirements.txt con los requerimientos del programa:

None

```
django>= 4.2  ## Aca se indica la version de django
psycob2-bynary ##El driver de PostgreSQL para Python
python-dotenv  ## Carga variables desde un archivo .env al entorno del
proceso
```

Estos requerimientos serán usados en el archivo **Dockerfile** el cual tendrá la siguiente estructura final:

```
FROM python:3.12-slim
ENV PYTHONUNBUFFERED=1
WORKDIR /app
# Instalar PostgreSQL client (necesario para psycopg2)
RUN apt-get update && apt-get install -y postgresql-client && rm -rf /var/lib/apt/lists/*
# Instalar dependencias Python
COPY Proyecto/SAFE/requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
# Copiar proyecto
COPY Proyecto/SAFE/ .
EXPOSE 8000
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

Este código especifica la version de python a usar, el environment, instala PostgreSQL client y las dependencias de python, y finalmente copia la ruta de la carpeta donde se encuentra el proyecto y finalmente expone el puerto que vamos a usar en el apartado WEB , en este caso el puerto 8000 y con el comando **CMD** accedemos a la consola para ejecutar por medio de python el archivo manage.py (lo veremos mas adelante) y hacer un “runserver” al puerto especificado.

3. Ahora crearemos un archivo que almacene las variables de entorno, esto se hace con la finalidad de proteger información sensible como las credenciales para acceder a la base de datos (los valores mostrados son de ejemplo)

```
None
DB_USER= usuario1
DB_PASSWORD= nose
DB_NAME= BD_Tutorial
DB_HOST=db
DB_PORT= 9999
DB_SSLMODE=disable
DJANGO_SECRET_KEY= key
DJANGO_DEBUG=True
```

C. Ejecución del Hola Mundo y creación del proyecto

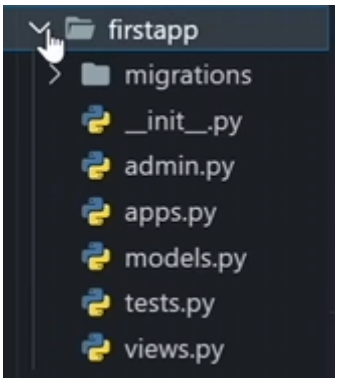
1. **Creación del proyecto:** Vamos a ejecutar los siguientes comandos en shell

```shell
django-admin startproject firstproject ##Esto crea el archivo manage.py
```

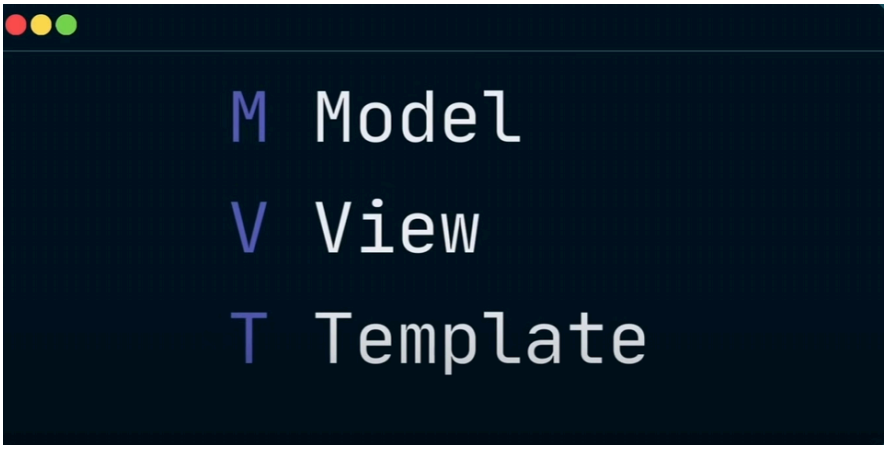
El siguiente código se ejecuta directamente en la terminal, el cual crea todos los archivos default del proyecto en django

```
python manage.py startapp firstapp ##Esto cre
```

Se debieron generar esta carpeta con estos archivos:



2. **Django Web Framework**



Django funciona para su apartado web por medio de 3 secciones, Model que representa los datos y su almacenamiento, View que representa las interacciones del usuario con la app y el apartado lógico y Template que representa la parte que “El usuario ve” es decir la UI.

3. **Models y conexión a la DB:** En el momento de crear el proyecto se genero automaticamente un archivo llamado Settings.py , vamos a abrirlo y a buscar el

apartado que dice **DATABASES =** En este apartado vamos a indicar a Django los parámetros para la base de datos de la siguiente manera:

```python
import os   ##Importamos os para poder acceder a los valores de las variables de entorno
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.getenv('DB_NAME'),
        'USER': os.getenv('DB_USER'),
        'PASSWORD': os.getenv('DB_PASSWORD'),
        'HOST': os.getenv('DB_HOST'),
        'PORT': os.getenv('DB_PORT'),
    }
}
```

A continuación vamos a crear el contenedor de la base de datos en Docker, esto se realizará usando los archivos .yml , .sql y .py que ya configuramos hasta el momento, usando los siguientes comandos en la consola:

```
docker compose up -d --build

docker compose exec -T db pg_isready -U postgres

docker compose exec -T web python manage.py migrate

docker compose exec -T db psql -U %DB_USER% -d %DB_NAME%< Proyecto/SAFE/db/init.sql

docker compose exec -T web bash -lc "python manage.py inspectdb > core/models.py"
```

El apartado **%DB_USER%** y **%DB_NAME%** se completan con los valores de el archivo .env
Estos comandos lo que hacen es:

1. Levantar los contenedores en Docker
2. Esperar a que POstgreSQL esté listo
3. Eealizar las migraciones de Django
4. Lee y realiza la ejecución de el archivo init.sql
5. Escanea la base de datos PostgreSQL , lee las tablas y genera automáticamente clases Python en models.py

Ya tenemos el archivo models.py el cual fue creado automáticamente al crear el proyecto. En este archivo se va a tener una “copia”de la estructura de la base de datos de la siguiente manera:

```
class Assignment(models.Model):
    type = models.TextField(blank=True, null=True)  # This field type is a guess.
    max_score = models.IntegerField(blank=True, null=True)
    due_date = models.DateTimeField(blank=True, null=True)

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

Aca se ve un ejemplo del “Modelado” de la tabla Assignment, el usuario no tiene que llenar este archivo ya que como se explicó anteriormente, en el apartado 5. de los comandos para montar el Docker, el comando #5 crea automáticamente estas clases.

4. Singleton para instanciar la DB: Una entidad Singleton es una entidad que está instanciada una única vez, en el contexto de una aplicación con base de datos es extremadamente útil ya que nos permite tener una única conexión con la base de datos, y así evitamos tener múltiples conexiones que puedan llegar a generar problemas.

Vamos a crear un archivo llamado singletondb.py y vamos a implementar 2 métodos: new y query

Python

```python
from django.db import connection

class DatabaseSingleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(DatabaseSingleton, cls).__new__(cls)
        return cls._instance

    def query(self, sql, params=None):
        with connection.cursor() as cursor:
            cursor.execute(sql, params or [])
            return cursor.fetchall()
```

a. **__new__** : Este método se llama al momento de instanciar a la clase DatabaseSingleton, en el cual si ya existe la instancia, devuelve esta misma y si no existe, la crea.
b. **query** : Como su nombre indica sirve para mandar una query a la base de datos, por medio de una clase de Django llamada connection la cual retorna la consulta en la base de datos

5. Views:

5.1 Al momento de crear el proyecto se creo un archivo views.py , en este archivo vamos a definir los métodos que se van a llamar durante la interacción de la UI con el usuario , además de esto se va a llamar una instancia del singletondb para poder acceder a los métodos para llamar a la base de datos. Pero antes de esto necesitamos llamar a una función que cree la UI de la aplicación, esta se llamara cuando el usuario haga una Request de abrir la aplicación.

```python
#1
from django.shortcuts import render, redirect
from django.views.decorators.http import require_POST
from ..singletondb import DatabaseSingleton

#2
db = DatabaseSingleton()

#3
def index(request):
    result = db.query("SELECT id, name, email FROM app_user ORDER BY id;")

    usuarios = []
    for r in result:
        usuarios.append({"id": r[0], "name": r[1], "email": r[2]})

    return render(request, "index.html", {"usuarios": usuarios})
```

Se recomienda que este archivo sea guardado en una carpeta a la misma altura sobre la que estan los demas archivos llamada Views y ahi guardar el archivo views.py

1. Las importaciones que se realizan son para renderizar la aplicación, redireccionar a un otra vista, recibir peticiones http y sobre nuestro archivo de singletondb
2. Instanciamos la clase DatabaseSingleton para acceder a sus métodos.
3. Esta función se encarga de renderizar y llamar a la interfaz gráfica en el archivo “index.html” de la siguiente forma:
    a. Realiza un Query a la db buscando el id , name , email de la tabla app_user

b. Itera sobre el resultado de esa Query y crea una lista de diccionarios llamada usuarios, en la cual en cada diccionario lo va llenando con la info de cada usuario.
c. Retorna el render sobre el html "**index.html"** y también devuelve la lista usuarios que creamos.

5.2 En la siguiente sección se explicará que se hace en "**index.html"** y la variable usuarios. Los siguientes 2 métodos son un ejemplo de métodos que se pueden crear en el archivo view los cuales serán llamados más adelante en la UI:

```python
@require_POST
def user_add(request):
    name = request.POST.get("name", "").strip()
    email = request.POST.get("email", "").strip()
    if name:
        db.query("INSERT INTO app_user (name, email) VALUES (%s, %s);",
[name, email or None])
    return redirect("index")

def user_del(request, pk):
    db.query("DELETE FROM app_user WHERE id = %s;", [pk])
    return redirect("index")
```

Estos 2 metodos agregan y eliminan a un usuario respectivamente de la base de datos, en el medoto **user_add** se llama a un metodo llamado **request.Post.get()** el cual recibe un request http y obtiene su valor, el resto de código se basa en hacer querys a la base de datos según la necesidad del método

5.3 Ahora vamos a crear un archivo llamado urls.py en la carpeta que se generó al crear la aplicación en el cual vamos a definir las urls de los métodos que creamos en views.py:

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

urlpatterns = [
    path("", index, name="index"),
    path("users/add/", user_add, name="user_add"),
    path("users/<int:pk>/del/", user_del, name="user_del"),
]
```

En este código importamos todos los archivos de la carpeta .views (dónde se guardó el archivo index.py) e importamos path de Django lo cual hace que cuando llegue cierta URL , llama a tal vista , por ejemplo el segundo path significa que cuando llegue la URL users/add/ , llama a la vista user_add la cual ya fue definida anteriormente en views.py

5.4 Ahora vamos a acceder a la carpeta que se generó al crear el proyecto , y vamos a acceder nuevamente a urls.py , notese que este archivo usted no tuvo que crearlo sino que ya venía definido por django. Lo que haremos en este archivo va a ser “linkear” las URL que definimos en el archivo anterior de la aplicación para que ahora las maneje internamente Django:

```python
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path("admin/", admin.site.urls),
    path("", include('firstapp.urls'))
]
```

El archivo es muy similar al anterior con la diferencia de que acá se define también la URL del admin de Django, y se importa **include** para llamar al archivo urls.py que creamos en la capa de la aplicación.

5.5 Finalmente tenemos que informar a la configuración de Django que se ha creado una aplicación y agregarla a la configuración del proyecto, esto lo vamos a realizar abriendo nuevamente settings.py y agregando al final del componente INSTALLED_APPS el nombre de la carpeta donde se encuentra nuestra aplicación de la siguiente manera:

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

6. Templates: Templates como su nombre en inglés lo dice son las plantillas para la UI , va a ser lo que el usuario va a ver en la aplicación, para esto vamos a crear un archivo **index.html** en el cual para este ejemplo queremos hacer el uso de agregar un usuario a la base de datos, y eliminar uno. Este archivo seguirá la estructura normal y básica de HTML la cual no se explicará en este tutorial

Agregar usuario a la base de datos:

```HTML
<form method="post" action="{% url 'user_add' %}">
	{% csrf_token %}
	<input name="name"  placeholder="Nombre" required>
	<input name="email" placeholder="Email (opcional)">
	<button>Agregar</button>
</form>
```

Este código utiliza la etiqueta <form> para crear un espacio de formulario, y aca podemos ver cómo la acción a realizar por el form es “user_add” y en las 2 líneas de abajo se configuran los espacios para poner la info del Form junto con el boton de ejecución llamado “Agregar”.

Mostrar usuarios y eliminar un usuario:

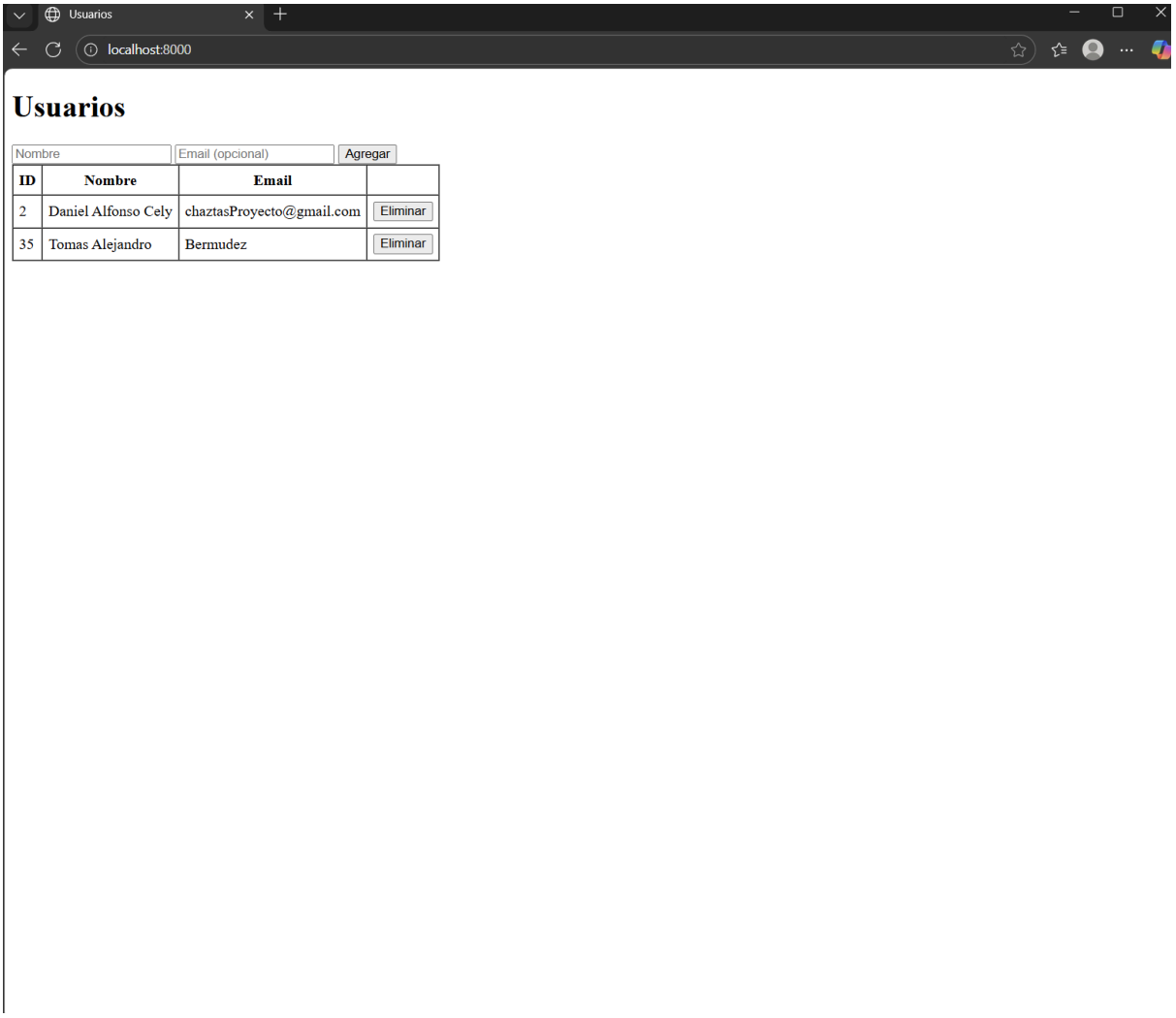
```HTML
<table border="1" cellpadding="6" cellspacing="0">
	<tr><th>ID</th><th>Nombre</th><th>Email</th><th></th></tr>
	{% for u in usuarios %}
	<tr>
		<td>{{ u.id }}</td>
		<td>{{ u.name }}</td>
		<td>{{ u.email }}</td>
		<td>
		<form method="post" action="{% url 'user_del' u.id %}"
style="display:inline">
			{% csrf_token %}
			<button onclick="return confirm('¿Eliminar
{{u.name}}?')">Eliminar</button>
		</form>
		</td>
	</tr>
	{% empty %}
	<tr><td colspan="4">Sin usuarios</td></tr>
	{% endfor %}
</table>
```

Ahora en este código utilizamos la etiqueta <table> la cual nos permite crear una tabla, recordemos que en index.py al realizar el render a index.html se paso como parámetro una lista de usuarios la cual va a ser sobre la que vamos a iterar para ir agregando los valores de cada usuario a la tabla, e internamente cada usuario va a tener una etiqueta form en la cual llamamos a la función user_del, para que cuando se ejecute el botón se realice la eliminación del usuario

7. Ejecución Hola mundo: Finalmente vamos a acceder a la aplicación web, lo primero que tenemos que hacer es activar el servidor por medio del siguiente comando:

```
None
python manage.py runserver
```

Finalmente vamos a abrir el link: localhost:8000



Se abrirá un panel como el siguiente, y así finaliza la ejecución de un Hola mundo usando el framework de Django.

Se pueden usar los métodos Agregar y Eliminar y comprobar que funcionen

Felicidades 😀 😀😀😀 has creado tu primera app utilizando Django, python, Docker y PostgreSQL !!!! Ahora te queda adaptar el resto de la aplicación a tu gusto y necesidades. 😀