Django Web Demo

Esta es una aplicrrollaación web desada con Django y explicada para que cualquiera que quiera aprender a desarrollar aplicaciones web con Python & Django pueda seguir esta guia y aprender a trabajar con este maravilloso framework.

Requisitos

Para empezar con este proyecto es necesario tener instalado Python y pip

Para instalar Python descargar el ejecutable y seguir los pasos
https://www.python.org/downloads/

Adicionalmente instalar pip, que es el gestor de paquetes de Python ejecutando el siguiente comando
`python get-pip.py`

Creación del entorno virtual

En primer lugar instalar el gestor de entornos mediante el administrador de paquetes
`pip install virtualenv`

Para crear el entorno virtual
`virtualenv web_env`

Esto creará el directorio `web_env`, el cual contiene el entorno virtual:

```
/web_env
  |__/Lib
  |__/Scripts
  |__.gitignore
  |__pyenv.cfg
```

El directorio `/Lib` contiene las librerías necesarias para correr nuestro código.
El directorio `/Scripts` contiene los ejecutables: como el intérprete de Python o pip.

Para listar todos los paquetes y/o librerías instalados en el entorno virtual
`pip freeze`
`pip list`

Para activar y desactivar el entorno ingresamos al directorio del entorno cd .\web_env\ y una vez dentro del directorio \web_env se puede activar o desactivar el entorno con los siguientes comandos.

- `.\Scripts\activate`
- `deactivate`

Instalación de Django

Para instalar Django ejecutar dentro del entorno virtual
`pip install django`

Podemos también comprobar la versión del framework con el comando
`django-admin --version`

Una vez instalado el framework con sus dependencias en el entorno virtual podemos crear el proyecto
`django-admin startproject webDemo`

Esto creará un directorio `.\webDemo` en el entorno virtual con la siguiente estructura:

```
webDemo/
  |__manage.py
  |__webDemo/
      |__ __init__.py
      |__ asgi.py
      |__ settings.py
      |__ urls.py
      |__ wsgi.py
```

- `webDemo/` es la carpeta del proyecto.
- `manage.py` es una utilidad para la linea de comandos (CLI) que permite interactuar con el proyecto de Django de varias formas. https://docs.djangoproject.com/en/1.8/ref/django-admin/
- El directorio interno `webDemo/` es el paquete de Python real para su proyecto. Su nombre es el nombre del paquete de Python que necesitará usar para importar cualquier cosa dentro de él (**Ej:** webDemo.urls).
- `__ init__.py` es un archivo vacío que le dice a Python que este directorio debe considerarse un paquete de Python.
- `settings.py` configuración para este proyecto de Django. https://docs.djangoproject.com/en/1.8/topics/settings/
- `urls.py` son las declaraciones de las URL's para este proyecto de Django. Puntos de entrada para la aplicación.
- `wsgi.py` es un punto de entrada para servidores web compatibles con WSGI para servir su proyecto.
- `asgi.py` además de WSGI, Django también admite la implementación en ASGI, el estándar emergente de Python para aplicaciones y servidores web asíncronos.

Formas de interactuar con el framework

```
$ django-admin <command> [options]
```

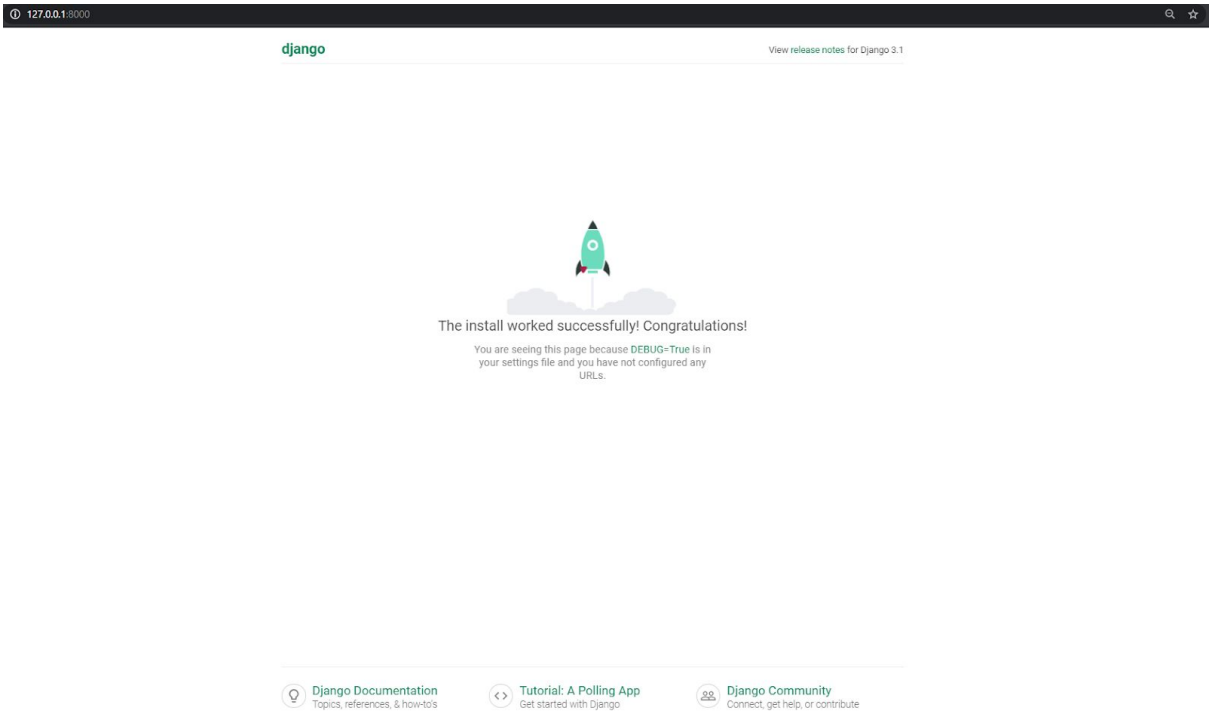
```
$ python manage.py <command> [options]
```

Arquitectura y Estructura

Para comprobar que el proyecto funciona dentro del directorio del proyecto `webDemo/` ejecutamos el siguiente comando para iniciar el servidor local de nuestro proyecto.

```
python manage.py runserver
```

Starting development server at http://127.0.0.1:8000/



Django viene con una utilidad que genera automáticamente la estructura de directorios básica de una aplicación, por lo que puede concentrarse en escribir código en lugar de crear directorios.

Nota: Proyectos vs. Aplicaciones
Una aplicación es una aplicación web que hace algo, por ejemplo, un sistema de registro web, una base de datos de registros públicos o una aplicación de encuesta simple. Un proyecto es una colección de configuraciones y aplicaciones para un sitio web en particular. Un proyecto puede contener varias aplicaciones. Una aplicación puede estar en varios proyectos.

Para crear una aplicación, hay que asegurarse de estar en el mismo directorio que manage.py y ejecutar el siguiente comando:

```
python manage.py startapp restaurant
```

Eso creará un directorio `\restaurant`, que tiene la siguiente estructura:

```
restaurant/
  |__ migrations/
     |__ _init_.py
  |__ __init__.py
  |__ admin.py
  |__ apps.py
  |__models.py
  |__tests.py
  |__views.py
```

Una vez creada la aplicación se debe agregar a la lista de aplicaciones instaladas en el archivo `webDemo/settings.py` de la siguiente manera:

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'restaurant',

)
```

Primeros pasos

Crear la primera vista con Django en el archivo `views.py` dentro del directorio de la aplicación `restaurant`

```
from django.shortcuts import render
from django.http import HttpResponse

# Create your views here.
def index(request):
    return HttpResponse("Hello, World!")
```

Ahora bien, para poder acceder a esta vista desde el navegador se debe configurar la ruta de entrada a esa vista en un nuevo archivo de Python que se crea de la siguiente manera: Damos click derecho sobre la carpeta de la aplicación ```\restaurant``` , New -> Python File, especificamos el nombre ```urls.py``` y declaramos las variables así:

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

urlpatterns = [
```

```
    path('', views.index, name='index')
]
```

Y por último, debemos enlazar el archivo restaurant\urls.py de la aplicación con el archivo webDemo\urls.py del proyecto.

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

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

Si entramos a la ruta http://127.0.0.1:8000/webDemo/ podemos ver nuestro primer Hello World en Django :)

Luego de crear nuestra primera vista debemos definir el modelo de nuestra aplicación dentro del archivo restaurant\models.py

```
from django.db import models
# Create your models here.

class Client(models.Model):
    name = models.CharField(max_length=50)
    phone = models.IntegerField()
    email =  models.EmailField(max_length = 50)

    def __str__(self):
        return
f"id={self.id},name={self.name},phone={self.phone},email={self.email}"

class Product(models.Model):
    name = models.CharField(max_length=50)
    price = models.FloatField()
    description = models.CharField(max_length=200)

    def __str__(self):
        return
f"id={self.id},name={self.name},price={self.price},description={self.de
scription}"

class Order(models.Model):
```

```
    client = models.ForeignKey(Client, on_delete=models.CASCADE)
    product = models.ManyToManyField(Product)

    def __str__(self):
        return f"id={self.id}, client={self.client}"
```

Una vez tenemos definido el modelo se puede ejecutar el siguiente comando para crear las migraciones que se plasmarán en el modelo en la base de datos

```
python manage.py makemigrations restaurant
```

Habiendo creado las migraciones se puede proceder a migrar el modelo que definimos a la base de datos mediante el siguiente comando

```
python manage.py migrate
```

Lo que creará un nuevo archivo en la carpeta `restaurant\migrations` con el historial de las migraciones realizadas.

Si quisiéramos obtener más información o cambiar la base de datos del proyecto (SQLite) podemos configurar esto en el archivo `settings.py` del proyecto.

```
# Database
# https://docs.djangoproject.com/en/3.1/ref/settings/#databases

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

Añadir y consultar registros desde el shell de Django

Ahora bien, para agregar datos a la base de datos podemos acceder al shell de Python mediante el siguiente comando

```
python manage.py shell
```

Una Vez dentro del shell de Python

```
>>> from restaurant.models import Client, Product, Order
```

```
>>> product = Product(name="Pollo en Salsa de Mango al curry",price=45.500,description="delicioso plato para compartir entre 2 personas")
```

```
>>> product.save()

>>> product.id

1

>>> product.name

'Pollo en Salsa de Mango al curry'

>>> Product.objects.all()

<QuerySet [<Product: id=2,name=Pollo en Salsa de Mango al curry,price=45.5,description=delicioso plato para compartir entre 2 personas>, <Product: id=3,name=Pollo en Salsa de Mango al curry,price=45.5,description=delicioso plato para compartir entre 2 personas>]>

>>> exit()
```

Para consultar la relación:

```
>>> Order.objects.first().product.all()

<QuerySet [<Product: id=2,name=Pollo en Salsa de Mango al curry,price=45.5,description=delicioso plato para compartir entre 2 personas>, <Product: id=4,name=Sopa

de Mariscos,price=40.0,description=Exquisita sopa de mariscos perfecta para el frio>]>
```

Utilizando la aplicación de administración que trae integrado el framework para facilitar el trabajo de crear, leer, actualizar y eliminar (CRUD) registros en la base de datos.

Para usar la aplicación de superusuario se deben importar los modelos en el archivo admin.py de la aplicación y registrarlos

```
from django.contrib import admin
# Register your models here.
from .models import Client,Product,Order
admin.site.register(Client)
admin.site.register(Product)
admin.site.register(Order)
```

Posteriormente crear un superusuario y ya estará disponible

```
python manage.py createsuperuser
```

```
Username: admin
```

Email address: example@gmail.com
Password: 123456
Password (again): 123456

El email que se muestra no es un email válido pero se recuerda que debe serlo para poder continuar.

Iniciamos el servidor local nuevamente con el comando python manage.py runserver
Ingresamos a la ruta que nos provee django por defecto que es
http://127.0.0.1:8000/admin/

Donde veremos un menú lateral con nuestras clases del modelo y las opciones a agregar (add) o cambiar (change)

Ahora que tenemos un modelo completo podemos empezar a hacer vistas más complejas

```
from django.shortcuts import render
from django.http import HttpResponse
from .models import Order

# Create your views here.
def index(request):
    orders = Order.objects.all()
    context = {
        'titulo_pagina': 'Ordenes del Restaurante',
        'listado_ordenes': orders
    }
    return render(request, 'orders.html' ,context)
```

Primero traemos la información de las órdenes de la base de datos, en el contexto especificamos las variables que se usarán en el template y el 'orders.html' es la plantilla que se renderiza al llamar a esta vista mediante la urls.

Para crear el template primero se debe añadir la siguiente configuracion en el archivo settings.py

```
STATICFILES_DIRS = [BASE_DIR / "static"]
```

Y ahora creamos el directorio donde almacenaremos los archivos estáticos de la aplicación, damos click derecho sobre la aplicación restaurant y creamos las carpetas static y templates. Dentro de la carpeta static creamos otra llamada css y posteriormente un archivo llamado styles.css. Dentro incluiremos los siguientes estilos:

```
body{
   font-family: 'Roboto', sans-serif;
```

```
    margin: 0;
}

h1,h2,h3{
    margin: 0;
}

h1{
    display: block;
    text-align:center;
    padding: 20px;
    color: white;
    background: #264653;
}

.contenedor{
    max-width: 90%;
    margin: 50px auto 0 auto;
}

h2{
    display: block;
    text-align:center;
    padding: 20px;
    color: black;
    border-radius: 10px;
    background: #f4a261;
}

ul{
    margin: 25px 0 0 0;
    list-style: none;
    padding: 0;
}

li > h3 {
    display: block;
    text-align:center;
    padding: 5px;
    color: black;
    border-radius: 10px;
```

```
    background: #2a9d8f;
}
```

Dentro de la carpeta templates creamos un archivo llamado base.html

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <link rel="preconnect" href="https://fonts.gstatic.com">
    <link
href="https://fonts.googleapis.com/css2?family=Roboto&display=swap"
rel="stylesheet">
    <link rel="stylesheet" href="{% static 'css/styles.css' %}">
    <title>Restaurante</title>
</head>
<style>

</style>
    <body>
        <h1>Aplicación Restaurante</h1>
        <div class="contenedor">

            <h2>{{titulo_pagina}}</h2>

            {% block contenido %} {% endblock %}
        </div>
    </body>
</html>
```

Y por último el archivo referente a la vista orders.html

```
{% extends 'base.html' %}

{% block contenido %}
<ul>
    {% for orden in listado_ordenes %}
    <li>
        <h3>Orden Id: {{orden.id}} </h3>
        <p><strong>El cliente es:</strong></p>
        <p>{{orden.client}} </p>
```

```
        <p><strong>Los productos que ordenó son:</strong></p>

        {% for producto in orden.product.all %}
        <p>{{producto}}</p>
        {% endfor %}
    </li>
    <hr>
    {% endfor %}
</ul>

{% endblock %}
```

Deberíamos terminar con la siguiente estructura.

```
restaurant/
    |__ migrations/
        |__ _init_.py
        |__ 0001_initial.py
    |__ static/
        |__ css/
            |__ styles.css
    |__ templates/
        |__ base.html
        |__ orders.html
    |__ __init__.py
    |__ admin.py
    |__ apps.py
    |__ models.py
    |__ tests.py
    |__ views.py
```

API REST

Se realizó la inclusión de la aplicación "apirest" dentro del proyecto "Django_Web_Demo" en la que se incluirá la exposición de servicios de API comportándose como backend y siendo consumida por el framework progresivo visto en clase, Vue en su versión 3.0.

Una vez creada la aplicación, la agregaremos al proyecto WebDemo mediante el archivo settings.py incluyendo la siguiente linea el apartado de INSTALLED_APPS quedando de la siguiente manera:

```
INSTALLED_APPS = [
```

```
  'django.contrib.admin',
  'django.contrib.auth',
  'django.contrib.contenttypes',
  'django.contrib.sessions',
  'django.contrib.messages',
  'corsheaders',
  'django.contrib.staticfiles',
  'restaurant',
  'apirest'
]
```

Añadimos nuestra nueva url de la aplicación para que esta pueda ser reconocida en el proyecto mediante el archivo urls.py dentro de la misma carpeta WebDemo en la sección de urlpatterns con la siguientes líneas:

```
urlpatterns = [
  path('admin/', admin.site.urls),
  path('webDemo/', include('restaurant.urls')),
  path('api/', include('apirest.urls'))
]
```

Agregaremos en el archivo `models.py` ubicado en la carpeta de nuestra aplicación `apirest`. Las siguientes líneas se incluirán con el fin de introducir nuestro modelo, en este caso, enfocado a restaurantes.

```
class Restaurant(models.Model):
  name = models.CharField(max_length=50)
  phone = models.CharField(max_length=15)
  email = models.CharField(max_length=80)
  website = models.CharField(max_length=100)
  iconUrl = models.CharField(max_length=120)
```

Ingresaremos en el archivo views.py en el que se expondrán nuestros servicios. En este caso, crearemos dos peticiones de tipo GET que nos permitan acceder a la

lista de restaurantes, a un restaurante exactamente y el filtrado de búsqueda de coincidencias por nombre:

```
from django.views import View
from .models import Restaurant
from django.http import JsonResponse
from django.forms.models import model_to_dict


class RestaurantListView(View):
  def get(self, request):
    if('name' in request.GET):
      restaurantList =
Restaurant.objects.filter(name__contains=request.GET['name'])
    else:
      restaurantList = Restaurant.objects.all()
    return JsonResponse(list(restaurantList.values()), safe=False)


class RestaurantDetailView(View):
  def get(self, request, pk):
    restaurant = Restaurant.objects.get(pk=pk)
    return JsonResponse(model_to_dict(restaurant))
```

Para poder visualizar los restaurantes en su formato JSON debemos asignarles un endpoint que permita identificar la petición que será realizada, por lo que nos dirigiremos al archivo urls.py y agregaremos las urls correspondientes:

```
from django.urls import path
from .views import RestaurantListView
from .views import RestaurantDetailView

urlpatterns = [
  path('restaurant/', RestaurantListView.as_view(), name='restaurant_list'),
```

```
  path('restaurant/<int:pk>/', RestaurantDetailView.as_view(),
name='restaurant')
]
```

Listo! Ya tenemos todo para probar nuestra API. Finalmente para poder interactuar con esta, debemos agregar el admin de Django para el control de nuestros datos. Lo realizaremos introduciendo las siguientes lineas en el archivo `admin.py` y nos fijaremos que estamos registrando el nombre de la clase almacenada en el modelo, en este caso, "Restaurant"

```
from django.contrib import admin
from .models import Restaurant

# Register your models here.
admin.site.register(Restaurant)
```

Repetimos el procedimiento para crear las migraciones que se plasmarán en el modelo en la base de datos:
`python3 manage.py makemigrations`

Una vez más, agregamos las migraciones a la base de datos mediante el siguiente comando:
`python3 manage.py migrate`

Con esto, ya podremos ejecutar nuestra API
`python3 manage.py runserver`

⚠️ Errores y soluciones

Si tu proyecto no se ejecuta correctamente:

- Recuerda encontrarte en el path correcto.
- Tener en un funcionamiento tu ambiente de desarrollo
- Ejecutar los comandos de migraciones al realizar alguna modificación que no permita el funcionamiento directo mediante `python manage.py runserver`.
- En caso de que no reconozca la palabra `python` en la consola, recuerda revisar hacia donde está apuntando tu IDE para el funcionamiento del ambiente de desarrollo. Podrías intentar en lugar de `python` la palabra reservada `python3` ó `py`. Por ejemplo `py manage.py runserver`

Al realizar el front bajo el framework progresivo Vue 3, este mostraba error al llamar mediante el axios cualquier petición perteneciente a la api del proyecto en Django:

```
No Access-Control-Allow-Origin header is present on the requested resource.
```

Puedes obtener más información de este error aquí:

- https://developer.mozilla.org/es/docs/Web/HTTP/CORS/Errors/CORSMissingAllowOrigin

Si también te sucede, debes seguir los siguientes pasos para solucionar los problemas de CORS:

Nos situamos en la consola donde estamos ejecutando nuestro proyecto de Django y colocaremos la siguiente línea:

```
python -m pip install django-cors-headers
```

En caso de no tener pip: (En Linux)

```
apt install python-pip
```

Luego, nos situaremos en el archivo `webDemo/settings.py` del proyecto `WebDemo` y nos iremos a la sección de `INSTALLED APPS` para agregar las siguientes líneas al final:

```
INSTALLED_APPS = [
....
  'corsheaders',
...
]
```

Asegurate de agregar la coma al final en caso de que no lo hayas puesto al final o podría obtener un ModuleNotFoundError. También deberás agregar una clase de middleware para escuchar las respuestas:

```
MIDDLEWARE = [
```

```
....
'corsheaders.middleware.CorsMiddleware',
   'django.middleware.common.CommonMiddleware',
...
]
```

CorsMiddleware debe colocarse lo más alto posible, especialmente antes de cualquier middleware que pueda generar respuestas como CommonMiddleware de Django o WhiteNoiseMiddleware de Whitenoise. Si no es antes, no podrá agregar los encabezados CORS a estas respuestas. Además, si estás utilizando CORS_REPLACE_HTTPS_REFERER, debe colocarse antes de CsrfViewMiddleware de Django.

Listo! Ya casi estamos. Debemos ingresar abajo de la sección `MIDDLEWARE` la línea que permitirá la excepción al problema de CORS:

```
CORS_ALLOW_ALL_ORIGINS = True
```

Después de ejecutar la aplicación en Django con éxito, solo nos queda ejecutar Vue para realizar la petición y ver que esta se ha realizado correctamente.