Servidor Básico con Django

Explicando el código

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

- Lo primero que hemos hecho es añadir en INSTALLED_APPS las aplicaciones de pokemon, ability y rest_framwork. Las dos primeras ya que son con las dos que vamos a trabajar y en las que vamos a crear los modelos, y la última porque la utilizaremos en el futuro.
- Todo esto es en el archivo de settings de nuestra carpeta servidor_basico_django, ya que es así como se llama nuestro proyecto.

```
from django.db import models

class Ability(models.Model):
    aName = models.CharField(max_length=30)
    description = models.CharField(max_length=100)
```

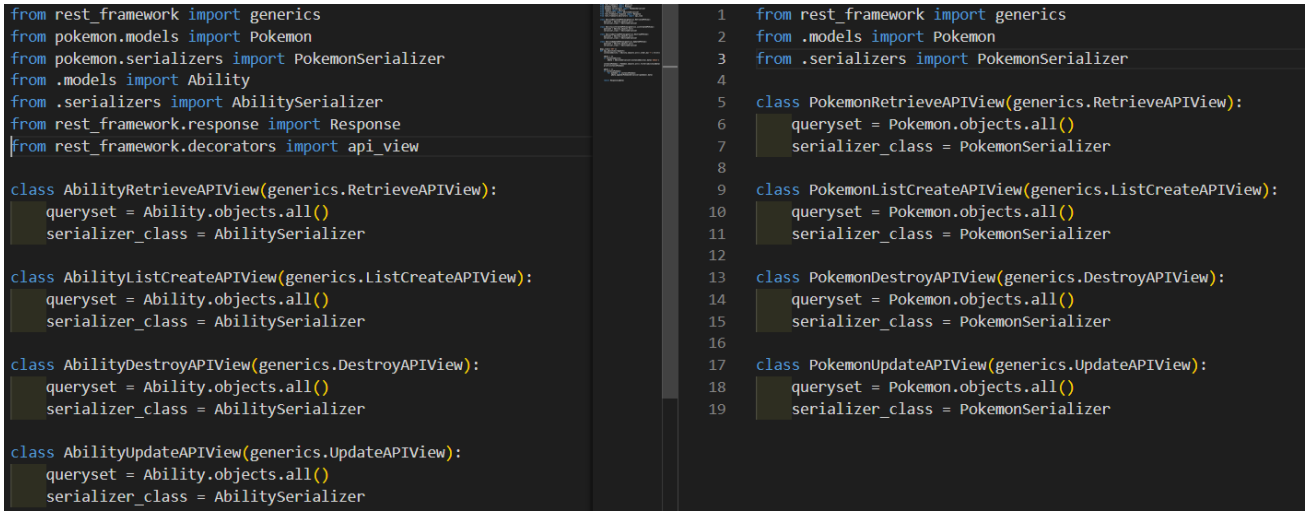
```
from django.db import models

class Pokemon(models.Model):
    dexNumber = models.IntegerField()
    name = models.CharField(max_length=15)
    ability = models.CharField(max_length=30)
```

- Después, creamos los modelos de ambas clases con sus respectivos atributos (en este tipo de screenshots siempre estará ability a la izquierda y pokemon a la derecha).

```
from rest_framework import serializers

from .models import Ability

class AbilitySerializer(serializers.ModelSerializer):
    class Meta:
        model = Ability
        fields = (
            'aName',
            'description'
        )
```

```
from rest_framework import serializers

from .models import Pokemon

class PokemonSerializer(serializers.ModelSerializer):
    class Meta:
        model = Pokemon
        fields = (
            'dexNumber',
            'name',
            'ability'
        )
```

- Tenemos que crear el archivo serializers en cada una de las apps, el cual utiliza el modelo para crear un serializer usando rest_framework los cuales usaremos más adelante para poder crear las vistas.



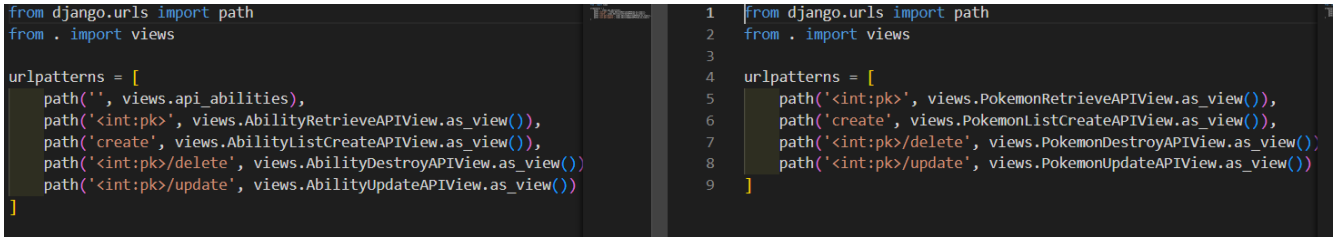
```python
from rest_framework import generics
from pokemon.models import Pokemon
from pokemon.serializers import PokemonSerializer
from .models import Ability
from .serializers import AbilitySerializer
from rest_framework.response import Response
from rest_framework.decorators import api_view

class AbilityRetrieveAPIView(generics.RetrieveAPIView):
    queryset = Ability.objects.all()
    serializer_class = AbilitySerializer

class AbilityListCreateAPIView(generics.ListCreateAPIView):
    queryset = Ability.objects.all()
    serializer_class = AbilitySerializer

class AbilityDestroyAPIView(generics.DestroyAPIView):
    queryset = Ability.objects.all()
    serializer_class = AbilitySerializer

class AbilityUpdateAPIView(generics.UpdateAPIView):
    queryset = Ability.objects.all()
    serializer_class = AbilitySerializer
```

```python
from rest_framework import generics
from .models import Pokemon
from .serializers import PokemonSerializer

class PokemonRetrieveAPIView(generics.RetrieveAPIView):
    queryset = Pokemon.objects.all()
    serializer_class = PokemonSerializer

class PokemonListCreateAPIView(generics.ListCreateAPIView):
    queryset = Pokemon.objects.all()
    serializer_class = PokemonSerializer

class PokemonDestroyAPIView(generics.DestroyAPIView):
    queryset = Pokemon.objects.all()
    serializer_class = PokemonSerializer

class PokemonUpdateAPIView(generics.UpdateAPIView):
    queryset = Pokemon.objects.all()
    serializer_class = PokemonSerializer
```

- Como decíamos antes, ahora usaremos los serializers para crear las vistas de cada app, y como nos solicitaban 4 genéricas de cada una hemos creado RetrieveAPIView (para mostrar uno de los objetos), ListCreateAPIView (para mostrarnos un listado de los objetos y poder crearlos), DestroyAPIView (para poder eliminar objetos) y UpdateAPIView (para poder modificarlos).

```python
@api_view(['GET'])
def api_abilities(request):
    instanceAbility = Ability.objects.all().order_by('?').first()

    aData = {}
    if instanceAbility:
        aData = AbilitySerializer(instanceAbility).data['aName']

    instancePokemon = Pokemon.objects.all().filter(ability=aData)

    pData = []
    if instancePokemon:
        for pokemon in instancePokemon:
            pData.append(PokemonSerializer(pokemon).data)


    return Response(pData)
```

- En el caso del archivo views de ability, vamos a crear tambien un api_view ya que queremos buscar todos los pokemon que tengan una ability random y la definimos como GET.
- Creamos el instanceAbility que es la variable en la que se almacena la primera ability obtenida de manera aleatoria, y si contiene datos, almacenamos en aData el aName, o nombre, de la ability que hemos obtenido.
- Con ese dato queremos almacenar en la variable instancePokemon todos los pokemon cuya ability encaje con el nombre de la ability obtenida, y si contiene datos, recorremos los datos y los vamos añadiendo en pData, el cual retornamos como respuesta.
- Ahora necesitamos crear el archivo urls en el que vamos a concretar las urls en las que funcionarán cada uno de los views.



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

urlpatterns = [
    path('', views.api_abilities),
    path('<int:pk>', views.AbilityRetrieveAPIView.as_view()),
    path('create', views.AbilityListCreateAPIView.as_view()),
    path('<int:pk>/delete', views.AbilityDestroyAPIView.as_view()),
    path('<int:pk>/update', views.AbilityUpdateAPIView.as_view())
]
```

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

urlpatterns = [
    path('<int:pk>', views.PokemonRetrieveAPIView.as_view()),
    path('create', views.PokemonListCreateAPIView.as_view()),
    path('<int:pk>/delete', views.PokemonDestroyAPIView.as_view()),
    path('<int:pk>/update', views.PokemonUpdateAPIView.as_view())
]
```

- En cada uno de los archivos creamos en urlpatterns los paths que queremos que sigan cada una de nuestras views.
- Para los RetrieveAPIView hemos indicado que debe ser un número, el cual seguirá al que corresponde con la pk de uno de los objetos. Está url será variable para cada uno de los que dispongamos.
- Para ListCreateAPIView hemos indicado que sea create.
- Para los DestroyAPIView y UpdateAPIView hemos indicado que debe ser un número, cómo en Retrieve, pero esta vez seguido de la acción que queremos realizar, delete para Destroy y update para Update.
- Para el urls de ability también hemos dejado la url vacía para poder lanzar el api_abilities, que es el api_view creado anteriormente.
- En este momento es cuando hacemos el makemigrations y migrate para que se actualicen los cambios y los modelos de nuestro proyecto.

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

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

- Por último, hay otro archivo urls, pero esta vez en la carpeta principal servidor_basico_django que debemos modificar.
- Ya que hemos establecido las mismas rutas para los ability y los pokemon, nos faltaba diferenciarlos y aquí lo hacemos indicando que antes de cada una de las rutas se debe poner pokemon/ o ability/ según la acción que queramos realizar. Cada uno de ellos apuntas a los path de sus respectivos archivos urls (es decir, para crear un nuevo pokemon habrá que ejecutarlo en la ruta localhost:8000/pokemon/create, y así respectivamente).
- Por último, se crea una nueva carpeta clients con el archivo test en el que vamos a realizar la introducción de los datos y vamos a realizar las pruebas que precisemos para probar que nuestras views están funcionando correctamente (podrás ver que tengo varias cosas comentadas, que son las rewuests que he utilizado para probar los métodos).

Conclusiones y observaciones

He podido observar que al borrar uno de los pokemon, en este caso Venusaur, que era el pokemon 5, luego el espacio de pokemon 5 no es rellenado con ningún otro pokemon y se queda vacío, quizás pueda ser porque va contando los pokemon que va creando y ese sería el pokemon 5 que se creó.

También me hubiera gustado utilizar como pk de cada pokemon su dexNumber, ya que es único para cada uno y hubiera sido algo bastante acertado, pero no he sabido como poder establecer uno de los atributos para que realizara esa tarea.

Este ejercicio me ha ayudado a ver las ideas más claras con django y sobre todo python ya que es un lenguaje que siempre me ha resultado algo complejo de entender para mí.