TEXT CLASS REVIEW

TEMAS A TRATAR EN EL CUE

- Entornos virtuales.
- Generar un archivo requirements.txt.
- Utiliza variables de entorno.
- El orden de los Imports.
- Modelos.
- Campo Booleano (BooleanField).
- Los Modelos Abstractos.
- La estructura de la plantilla (Templates).

MEJORES PRÁCTICAS EN DJANGO

Al iniciar un proyecto de software, y comenzar a escribir código, pensamos que existen mejores formas de realizar alguna funcionalidad; en este sentido, existen algunas buenas prácticas que se implementan cuando se trabaja con Django.

ENTORNOS VIRTUALES

De preferencia, es indispensable utilizar Entornos Virtuales para cada proyecto. No se recomienda utilizar un mismo entorno para más proyectos, esto es motivado a que cada proyecto no siempre utiliza las mismas versiones de las librerías, por lo que instalarlas de forma global en tu sistema operativo, sería llenar de librerías y temporales que generan lentitud en el funcionamiento del computador, pues cada vez que necesites cambiar de versión deberás desinstalar la existente e instalar la necesaria.

Es por ello que se pueden crear entornos virtuales de diversas maneras. Hay varias librerías que permiten esto, desde: `virtualenv`, `conda`, `venv`, `pipenv`, `virtualenvwrapper`, entre otros.

GENERAR UN ARCHIVO REQUIREMENTS.TXT

Al iniciar y crear un proyecto, y si deseamos colocarlo en un repositorio público o para un equipo de trabajo, debemos proporcionar las librerías y las respectivas versiones de las mismas que utiliza el proyecto. Por ello podemos generar un archivo donde indiquemos las librerías o paquetes requeridos, todo esto por convención, y este archivo suele denominarse requirements.

Para crear este archivo se utiliza el comando:

```
pip freeze > requirements.txt
```

Generando un archivo con los nombres de las librerías y sus versiones. Para instalar las dependencias necesarias, se utiliza el comando:

```
pip install -r requirements.txt
```

UTILIZAR VARIABLES DE ENTORNO

Para separar nuestro archivo de configuración o settings, es una práctica muy recomendada, pues la información sensible la pueden tener todos los desarrolladores. Por ello, algunos equipos optan por utilizar las denominadas Variables de Entorno, y de esta forma, los miembros sólo tendrán las credenciales de desarrollo y nada de producción, generando seguridad y una separación de las funcionalidades dependiendo el entorno de desarrollo.

Se recomienda usar `django-environ`, que es un paquete de Python que permite usar la metodología de doce factores para configurar su aplicación Django con variables de entorno. Se puede hacer referencia en: https://pypi.org/project/django-environ/

EL ORDEN DE LOS IMPORTS

Las importaciones son una parte inevitable del desarrollo de Python y Django. Pep8, que es la guía de estilo oficial de Python, recomienda que las importaciones se coloquen en la parte superior del archivo, en líneas separadas, y que se agrupen en el siguiente orden:

- `Future` (si es que se desea importar).
- Librerías nativas o estándares de Python.
- Librerías de terceros.
- Paquetes propios de Django, o el framework a utilizar.
- Paquetes o aplicaciones locales (de nuestro proyecto).
- `Try/Except` en un import.

Aquí hay un archivo de ejemplo `views.py` de una aplicación. Por lo general, cuando escribimos los imports en nuestro proyecto se puede cometer una ligera mala práctica, que es importar todos los paquetes que necesitemos en desorden. Se recomienda seguir el siguiente orden:

```python
# future
from __future__ import unicode_literals

# standard library
import json
from itertools import chain

# third-party
import bcrypt

# Django
from django.http import Http404
from django.http.response import (
    Http404, HttpResponse, HttpResponseNotAllowed, StreamingHttpResponse,
    cookie,
)


from .models import LogEntry

# try/except
try:
    import yaml
except ImportError:
```

```
    yaml = None

CONSTANT = 'foo'


class Example:
    # ...
```

MODELOS

Generalmente se recomienda que los modelos sean nombrados en forma Singular, y si contiene más de una palabra, la primera letra de cada palabra debe ir en Mayúscula.(`UpperCamelCase`).

```
class Owner(models.Model):
    pass

class Item(models.Model):
    pass
Custom tool bar! Text/images etc.
```

NOMBRE DE CAMPOS DE MODELO

Los nombres de los campos de un modelo se deben escribir en minúscula, y si el campo posee más de una palabra, deben unirse mediante subguiones(_).

```
# ESTO ES CORRECTO
class Person(models.Model):
    first_name = models.CharField(max_length=20)
    last_name = models.CharField(max_length=40)

# ESTO ES INCORRECTO
class Person(models.Model):
    FirstName = models.CharField(max_length=20)
    Last_Name = models.CharField(max_length=40)
```

ORDEN DE UN MODELO

Cuando escribimos nuestros modelos, debemos mantener un orden al escribir los campos, los campos de managers personalizados, la clase Meta, el método `__str__`, entre otros. Procura mantener el siguiente orden:

- Los campos que hagan referencia a `Managers` personalizados.
- Class `Meta`.
- `def __str__()`.
- `def save()`.
- `def get_absolute_url()`.
- Otros métodos propios que desees añadir.

```python
class Person(models.Model):
    first_name = models.CharField(max_length=20)
    last_name = models.CharField(max_length=40)

    custom_manager = CustomManager()

    class Meta:
        pass

    def __str__(self):
        pass

    def save():
        pass

    def get_absolute_url():
        pass


    def get_full_name():
        pass
```

CHOICES (OPCIONES) EN MODELO

Si se definen opciones para un campo de modelo determinado, defina cada opción como una lista de tuplas, con un nombre en mayúsculas como atributo de clase en el modelo.

```python
class MyModel(models.Model):
    DIRECTION_UP = 'U'
    DIRECTION_DOWN = 'D'
    DIRECTION_CHOICES = [
        (DIRECTION_UP, 'Up'),
        (DIRECTION_DOWN, 'Down'),
    ]
```

CAMPO BOOLEANO (BOOLEANFIELD)

Se recomienda no declarar un campo de tipo BooleanField con `null=True` o `blank=True`; si desea que el campo booleano sea opcional, se utiliza el tipo de campo `NullBoleanField`. También es importante no usar muchos Boolean si no es necesario. Por lo general, utilizamos campos Boolean para manejar Estados del modelo, es decir, las diferentes etapas que puede tener cada instancia de este. Por ejemplo:

```python
class Article(models.Model):
    is_published = models.BooleanField(default=False)
    is_verified = models.BooleanField(default=False)
    …
```

Sin embargo, para esto podemos manejar los estados del modelo por medio de opciones (`Choices`), teniendo en cuenta que dependerá del caso de uso o requerimiento, pero de ser posible, aplicarlo. Por ejemplo:

```python
class Article(models.Model):
    NEW = 'N'
    VERIFIED = 'F'
    PUBLISHED = 'P'
    STATUSES = [
        (NEW, 'new'),
        (VERIFIED, 'verified'),
        (PUBLISHED, 'published')
    ]

    status = models.IntegerField(choices=STATUSES)
```

LOS MODELOS ABSTRACTOS

Se recomienda, por lo general, el uso de modelos Abstractos para evitar repetir la escritura de campos globales. Recuerda que los Modelos no son más que clases, y se utiliza la Programación Orientada a Objetos para su definición, es decir, que podemos utilizar la Herencia o Abstracción. Por ejemplo:

```
class BaseModel(models.Model):
    state = models.BooleanField(default=True)
    created = models.DateField(auto_now_add=True, auto_now=False)
    modified = models.DateField(auto_now=True, auto_now_add=False)
    deleted = models.DateField(auto_now=True, auto_now_add=False)

    class Meta:
        abstract = True


class Comment(BaseModel):
    pass
```

LA ESTRUCTURA DE LA PLANTILLA (TEMPLATES)

Hay dos formas principales de organizar la estructura de su plantilla en Django: la forma predeterminada a nivel de aplicación, y un enfoque personalizado a nivel de proyecto.

1. **Nivel de aplicación**

   De forma predeterminada, el cargador de plantillas de Django buscará una templates carpeta dentro de cada aplicación. Pero para evitar problemas con el espacio de nombres, también debe repetir el nombre de la aplicación en una carpeta debajo de esa, antes de agregar su archivo de plantilla.

   Por ejemplo: si tuviéramos un `example_project` con una aplicación de `pages`, y un archivo home.html de plantilla, la estructura adecuada sería así:

   Dentro de la aplicación `pages` creamos un directorio templates, luego un directorio pages, y finalmente, nuestro archivo `home.html`.

```
├── example_project
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
│   └── pages
│       ├── __init__.py
│       ├── admin.py
│       ├── apps.py
│       ├── models.py
│       ├── tests.py
│       └── views.py
│       ├── templates
│           ├── pages
│               ├── home.html
└── manage.py
```

2. **Nivel de proyecto**

   A medida que los proyectos de Django aumentan de tamaño, algunas veces es más conveniente tener todas las plantillas en un solo lugar, en vez de tener que buscarlas en varias aplicaciones. Con un cambio de una sola línea en nuestro archivo `settings.py`, podemos hacer esto. Actualice la configuración `'DIRS'` de la variable `TEMPLATES` de la siguiente manera, que especifica que además de buscar un directorio de plantillas de nivel de aplicación, el cargador de plantillas de Django también debe buscar un directorio de plantillas de nivel de proyecto.

```
# settings.py
TEMPLATES = [
    {
        ...
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        ...
    },
]
```

   Luego, crea un directorio `templates` al mismo nivel que el proyecto. Aquí hay un ejemplo de cómo se vería con el archivo `home.html`:

```
├── example_project
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
│   └── pages
│       ├── __init__.py
│       ├── admin.py
│       ├── apps.py
│       ├── models.py
│       ├── tests.py
│       └── views.py
├── templates
│   ├── home.html
└── manage.py
```