Práctica 11: ORM de Django y QuerySets

En esta práctica aprenderás cómo Django se conecta a la base de datos y almacena los datos en ella.

¿Qué es un QuerySet?

Un QuerySet es, en esencia, una lista de objetos de un modelo determinado. Un QuerySet te permite leer los datos de una base de datos, filtrarlos y ordenarlos.

Django shell

Abre la consola (no la de PythonAnywhere) y escribe este comando:

```
(myvenv) ~/django-daw$ python manage.py shell
```

El resultado debería ser:

```
(InteractiveConsole)
>>>
```

Ahora estás en la consola interactiva de Django. Es como la consola de Python, pero con un toque de magia Django. Puedes utilizar todos los comandos Python aquí también, por supuesto.

Todos los objetos

Vamos a mostrar todos nuestros posts primero. Puedes hacerlo con el siguiente comando:

```
>>> Post.objects.all()
Traceback (most recent call last):
      File "<console>", line 1, in <module>
NameError: name 'Post' is not defined
```

Apareció un error. Nos dice que no hay ningún objeto Post. Esto es correcto, hizo falta que lo importáramos primero.

```
>>> from blog.models import Post
```

Esto es simple: importamos el modelo `Post` de `blog.models`. Vamos a intentar mostrar todos los posts nuevamente:

```
>>> Post.objects.all()
<QuerySet [<Post: my post title>, <Post: another post title>]>
```

Esta es una lista de los posts creados anteriormente. Hemos creado estos posts usando la interfaz del administrador de Django. Sin embargo, ahora queremos crear nuevos posts usando Python, veamos como se hace.

Crear un objeto

Esta es la forma de crear un nuevo objeto Post en la base de datos:

```
>>> Post.objects.create(author=me, title='Sample title', text='Test')
```

Pero hay un ingrediente faltante: `me`. Necesitamos pasar una instancia del modelo `User` como autor. Esto se logra de la siguiente manera.

Primero importamos el modelo User:

```
>>> from django.contrib.auth.models import User
```

Veamos que usuarios tenemos en nuestra base de datos, prueba esto:

```
>>> User.objects.all()
<QuerySet [<User: daw>]>
```

Este es el super usuario que creamos anteriormente, vamos a obtener una instancia de ese usuario ahora:

```
me = User.objects.get(username='daw')
```

Como puedes ver, hicimos un `get` de un `User` con el `username` que sea igual a 'daw'. Acuérdate de poner tu nombre de usuario para obtener tu usuario. Ahora finalmente podemos crear nuestro primer post:

```
>>> Post.objects.create(author=me, title='Sample title', text='Test')
```

Probemos si funcionó.

```
>>> Post.objects.all()
<QuerySet [<Post: my post title>, <Post: another post title>, <Post: Sample title>]>
```

En este punto deberías de haber visto un nuevo post en la lista.

Agrega más posts

Ahora puedes divertirte un poco y añadir más posts para ver cómo funciona. Añade 2 ó 3 más y avanza a la siguiente parte.

Filtrado de objetos

Una parte importante de los QuerySets es la habilidad para filtrarlos. Digamos que queremos encontrar todos los posts cuyo autor es el User 'daw'. Usaremos `filter` en vez de `all` en `Post.objects.all()`. En los paréntesis estableceremos qué condición(es) deben cumplirse por un post del blog para terminar en nuestro queryset. En nuestro caso sería `author` es igual a `me`. La forma de escribirlo en Django es: `author=me`. Ahora nuestro bloque de código se ve como esto:

```
>>> Post.objects.filter(author=me)
<QuerySet [<Post: Sample title>, <Post: Post number 2>, <Post: My 3rd post!>, <Post: 4th title of post>]>
```

¿O tal vez querramos ver todos los posts que contengan la palabra 'title' en el campo `title`?

```
>>> Post.objects.filter(title__contains='title')
<QuerySet [<Post: Sample title>, <Post: 4th title of post>]>
```

> Hay dos guiones bajos (`_`) entre `title` y `contains`. Django ORM utiliza esta sintaxis para separar los nombres de los campos ("title") y operaciones o filtros ("contains"). Si sólo utilizas un guión bajo, obtendrás un error como "FieldError: Cannot resolve keyword title_contains".

También puedes obtener una lista de todos los posts publicados. Lo hacemos filtrando los posts que tienen el campo `published_date` en el pasado:

```
>>> from django.utils import timezone
>>> Post.objects.filter(published_date__lte=timezone.now())
[]
```

Desafortunadamente, ninguno de nuestros posts han sido publicados todavía. Vamos a cambiar esto, primero obtén una instancia de un post que queramos publicar:

```
>>> post = Post.objects.get(title="Sample title")
```

Luego utiliza el método `publish` para publicarlo.

```
>>> post.publish()
```

Ahora intenta obtener la lista de posts publicados nuevamente (presiona la tecla con la flecha hacia arriba 3 veces y presiona `Enter`):

```
>>> Post.objects.filter(published_date__lte=timezone.now())
<QuerySet [<Post: Sample title>]>
```

Ordenando objetos

Los QuerySets también te permiten ordenar la lista de objetos. Intentemos ordenarlos por el campo `created_date`:

```
>>> Post.objects.order_by('created_date')
<QuerySet [<Post: Sample title>, <Post: Post number 2>, <Post: My 3rd post!>, <Post: 4th title of post>]>
```

También podemos invertir el ordenamiento agregando `-` al principio:

```
>>> Post.objects.order_by('-created_date')
<QuerySet [<Post: 4th title of post>,  <Post: My 3rd post!>, <Post: Post number 2>, <Post: Sample title>]>
```

Encadenando QuerySets

También puedes combinar QuerySets al **encadenarlos** juntos:

```
>>> Post.objects.filter(published_date__lte=timezone.now()).order_by('published_date')
```

Esto es muy poderoso y te permite crear búsquedas complejas.

Para cerrar la consola, escribe:

```
>>> exit()
$
```