TEXT CLASS REVIEW

TEMAS A TRATAR EN EL CUE:

- Recuperando registros
- Aplicando filtros en recuperación de registros
- Ejecutando queries SQL
- Mapeando campos de consultas al modelo

EJECUTANDO SENTENCIAS DE RECUPERACIÓN CON ORM

El ORM de Django es una implementación del concepto de mapeo de objeto relacional (ORM). Es una de las características más poderosas de Django, que le permite interactuar con su base de datos, como lo haría con instrucciones SQL (Structured Query Language).

RECUPERANDO REGISTROS

Para recuperar objetos de la base de datos, construimos un `QuerySet`, es decir una consulta a través de una clase `Manager`.

Un `QuerySet` representa una colección de objetos a partir de la base de datos. Que pueden tener cero, uno, o muchos filtros (`filters`) criterios que pueden reducir la colección basándose en parámetros dados. En términos SQL, un `QuerySet` equivale a una declaración `SELECT`, y un filtro es una cláusula de limitación `WHERE` o `LIMIT`.

Podemos traer un `QuerySet` usando el `Manager` de su modelo. Cada modelo tiene al menos un Manager, y este es llamado `objects` por `default`. Los Managers son accesibles únicamente a través de las clases del modelo, en vez que desde una instancia, para aplicar una separación entre las operaciones a “nivel de tabla”, y las operaciones “a nivel de registro”.

El `Manager` es la principal fuente de `QuerySets` de un modelo. Actúa como una clase “raíz” de `QuerySet`, que describe todos los objetos de la tabla de base de datos del modelo. Por ejemplo, `Blog.objects` es el `QuerySet` inicial que contiene todos los objetos Blog de la base de datos.

RECUPERANDO TODOS LOS REGISTROS DEL MODELO

Este es el caso más simple, solo tenemos que usar el método `all()` de nuestro modelo:

```
from contabilidad.models import Cliente, Factura
clientes = Cliente.objects.all()
for cliente in clientes:
print(f"Nombre: {cliente.nombre}, Apellidos: {cliente.apellidos}")
```

Y en el caso de los modelos con relaciones, también podemos acceder a los datos del modelo relacionado, usando el nombre del campo que tiene la relación, seguido de un doble guion bajo (__), y por último el campo del otro modelo:

```
from contabilidad.models import Cliente, Factura
facturas = Factura.objects.all()
for factura in facturas:
    print(f"Pagada: {factura.pagada}, Importe: {factura.importe}, RFC
Cliente: {factura.cliente__rfc}")
```

Como se puede observar al final de la última línea, para acceder al campo `rfc` desde el modelo `Factura`, usamos `factura.cliente__rfc`.

APLICANDO FILTROS EN RECUPERACIÓN DE REGISTROS

Podemos usar el método `filter()` como un símil del `WHERE` de SQL, y así obtener un conjunto de datos limitado por condiciones. Los parámetros de `filter()` corresponden a los nombres de las variables del modelo, y se pueden agregar modificadores con el sufijo __ modificador.

```
from contabilidad.models import Cliente, Factura
import datetime
# obtenemos exclusivamente a los clientes activos
clientes = Cliente.objects.filter(activo=True)
# buscamos a los clientes que su fecha de nacimiento > 1980-03-01
# y que no estén activos. El modificador '__gt' significa greater than, o
>
fecha = datetime.date(1980, 3, 1)
clientes=Cliente.objects.filter(activo=False,fecha_nacimiento__gt=fecha)
# si queremos un filtro inclusivo usamos '__gte' (greater than equal)
clientes=Cliente.objects.filter(activo=False,fecha_nacimiento__gte=fecha)
# para buscar un rango de fechas y todos los que se llamen Carlos:
inicio = datetime.date(1980, 3, 1)
final = datetime.date(2000, 1, 1)
clientes = Clientes.objects.filter(
    fecha_nacimiento__range=(inicio, final),
```

```
    nombre="Carlos")
# podemos buscar por los valores de una lista
clientes = Clientes.objects.filter(nombre__in=["Juan", "María",
"Gabriela"])
```

Cuando tenemos modelos con relaciones el procedimiento es muy similar, solo tenemos que usar __ para acceder al modelo referenciado, y si queremos usar un modificador, usamos el sufijo __ modificador.

Busquemos las facturas pagadas de los clientes nacidos en la década de los 80's:

```
from contabilidad.models import Cliente, Factura
import datetime
inicio = datetime.date(1980, 1, 1)
final = datetime.date(1989, 12, 31)
facturas = Factura.objects.filter(
    pagada=True,
    cliente__fecha_nacimiento__range=(inicio, final))
```

Se puede también restringir el número de registros regresados usando la notación de slices nativa de Python, y esto se traduce al equivalente `LIMIT` de SQL. Para regresar los primeros diez registros:

```
Factura.objects.all()[:10]
```

O si queremos regresar los registros del 5 al 20, usamos:

```
Factura.objects.all()[5:20]
```

EJECUTANDO QUERIES SQL

Django le ofrece dos formas de realizar consultas SQL sin formato: puede usar `Manager.raw()` para realizar consultas sin formato, y devolver instancias de modelos; o puede evitar la capa del modelo por completo, y ejecutar SQL personalizado directamente.

REALIZAR CONSULTAS EN BRUTO

El método de administrador `raw()` se puede utilizar para realizar consultas SQL sin formato que devuelven instancias de modelo:

```
Manager.raw(raw_query, params=(), translations=None)
```

Este método toma una consulta SQL sin procesar, la ejecuta y devuelve una instancia `django.db.models.query.RawQuerySet`. Esta instancia de `RawQuerySet` se puede iterar como un `QuerySet` normal para proporcionar instancias de objetos.

Esto se ilustra mejor con un ejemplo. Supongamos que se tiene el siguiente modelo:

```
class Person(models.Model):
    first_name = models.CharField(...)
    last_name = models.CharField(...)
    birth_date = models.DateField(...)
```

Se puede entonces ejecutar un SQL personalizado como este:

```
>>> for p in Person.objects.raw('SELECT * FROM myapp_person'):
...     print(p)
John Smith
Jane Jones
```

Este ejemplo es exactamente lo mismo que ejecutar `Person.objects.all()` del ORM de Django. Sin embargo, `raw()` tiene otras opciones que lo hacen muy poderoso.

MAPEANDO CAMPOS DE CONSULTAS AL MODELO

`raw()` asigna automáticamente campos en la consulta a campos en el modelo. El orden de los campos en su consulta no importa. En otras palabras, las dos consultas siguientes funcionan de manera idéntica:

```
Person.objects.raw('SELECT id, first_name, last_name, birth_date FROM
myapp_person')

Person.objects.raw('SELECT last_name, birth_date, first_name, id FROM
myapp_person')
```

La correspondencia se realiza por nombre. Esto significa que puede usar las cláusulas `AS` de SQL para asignar campos en la consulta para modelar campos. Entonces, si tuviera otra tabla que tuviera datos de `Person`, podría asignarse fácilmente a instancias de `Person`:

```
Person.objects.raw('''SELECT first AS first_name,
                          last AS last_name,
                          bd AS birth_date,
                          pk AS id,
                          FROM some_other_table''')
```

Mientras los nombres coincidan, las instancias del modelo se crearán correctamente. Alternativamente, puede asignar campos en la consulta para modelar campos usando el argumento de `translations` a `raw()`. Este es un diccionario que asigna nombres de campos en la consulta a nombres de campos en el modelo. Por ejemplo, la consulta anterior también podría escribirse:

```
name_map = {'first': 'first_name', 'last': 'last_name', 'bd': 'birth_date',
'pk': 'id'}

Person.objects.raw('SELECT * FROM some_other_table', translations=name_map)
```

BÚSQUEDAS DE ÍNDICE

`raw()` admite indexación, por lo que si solo necesita el primer resultado, puede escribir:

```
first_person = Person.objects.raw('SELECT * FROM myapp_person')[0]
```

Sin embargo, la indexación y el corte no se realizan a nivel de base de datos. Si tiene una gran cantidad de objetos `Person` en su base de datos, es más eficiente limitar la consulta en el nivel SQL:

```
first_person = Person.objects.raw('SELECT * FROM myapp_person LIMIT 1')[0]
```