HINTS

EJECUTANDO SENTENCIAS DE RECUPERACIÓN CON ORM.

En ocasiones, solo se necesita un registro en particular, y para esto usamos el método `get()`. Los parámetros de este método son los nombres de los campos, los cuales nos servirán como condiciones SQL para obtener el registro. El caso más común es filtrar por el id del registro, por lo que si queremos obtener al Cliente con `id` número 5, hacemos lo siguiente:

```
from contabilidad.models import Cliente
pedro = Cliente.objects.get(id=5)
print(pedro.nombre)
```

En este caso no obtenemos un `QuerySet`, pero sí un objeto con la información de nuestro registro.

> Lo más común es usar `get()` con el `id`, pero si creamos manualmente la llave primaria de nuestro modelo, le podemos poner el nombre que nosotros consideremos más adecuado. Para facilitarlo, Django pone a nuestra disposición la variable `pk`, que apunta a la llave primaria. Es por esto que se recomienda usar `pk` con `get(): Cliente.objects.get(pk=5)`.

A diferencia de `filter()` y `all()`, si `get()` no encuentra ningún registro, lanzará la excepción modelo.DoesNotExist, por lo que hay que manejarla siempre que usemos `get()`:

```
from contabilidad.models import Cliente
cliente_id=5
try:
    pedro = Cliente.objects.get(pk=cliente_id)
    print(pedro.nombre)
except Cliente.DoesNotExist:
    print(f"No existe el registro con id={cliente_id}")
```

Muchas veces necesitamos crear el usuario que estamos buscando si es que no existe, de esta forma:

```
from contabilidad.models import Cliente
import datetime
cliente_id=5
fecha_nacimiento = datetime.date(1980, 12, 5)
try:
    pedro = Cliente.objects.get(
        nombre="Pedro",
        apellidos="Aguilar Ramírez",
```

```
        rfc="AGRM-801205-111",
        fecha_nacimiento=fecha_nacimiento)
except Cliente.DoesNotExist:
    pedro = Cliente.objects.create(
        nombre="Pedro",
        apellidos="Aguilar Ramírez",
        rfc="AGRM-801205-111",
        fecha_nacimiento=fecha_nacimiento)
```

Además de engorroso, esto rompe con el principio DRY. Afortunadamente Django tiene un atajo:

```
from contabilidad.models import Cliente
import datetime
cliente_id=5
fecha_nacimiento = datetime.date(1980, 12, 5)
pedro = Cliente.objects.get_or_create(
    nombre="Pedro",
    apellidos="Aguilar Ramírez",
    rfc="AGRM-801205-111",
    fecha_nacimiento=fecha_nacimiento)
```

Solo tenemos que hacer la búsqueda usando todos los campos que no tengan valores `default`, para que el registro pueda ser creado si no existe.

Aunque no estemos trabajando directamente con SQL, al final Django convierte las consultas del ORM a código SQL. En ocasiones es muy útil inspeccionar este código para validar que nuestra consulta sea la correcta. Para ello solo tenemos que revisar el contenido de la variable query:

```
from contabilidad.models import Cliente
clientes=Cliente.objects.filter(apellidos__startswith="García").order_by
("apellidos", "-fecha_nacimiento")
print(clientes.query)
```

SENTENCIAS SQL

El ORM de Django es muy sencillo de usar, y potente al mismo tiempo. Sin embargo, habrá situaciones donde una consulta SQL no la podamos expresar con el ORM, pero Django nos permite ejecutar raw queries en estos casos.

Algunas veces, incluso `Manager.raw()` no es suficiente: es posible que deba realizar consultas que no se asignen limpiamente a los modelos, o ejecutar directamente consultas `UPDATE`, `INSERT`, o `DELETE`.

En estos casos, siempre se puede acceder directamente a la base de datos, enrutando por completo la capa del modelo.

El objeto `django.db.connection` representa la conexión de base de datos predeterminada. Para usarla, llame a `connection.cursor()` que facilita un objeto de cursor. Luego, llame a `cursor.execute(sql, [params])` para ejecutar el SQL, y `cursor.fetchone()` o `cursor.fetchall()` para devolver las filas resultantes, que se estudiarán con más detalle en el siguiente CUE.