TEXT CLASS REVIEW

TEMAS A TRATAR EN EL CUE:

- Exclusión de campos del modelo
- Añadiendo anotaciones
- Pasando parámetros a raw()
- Ejecutando SQL personalizado directamente
- Conexiones y cursores
- Invocación a procedimientos almacenados

EXCLUSIÓN DE CAMPOS DEL MODELO Y AÑADIENDO ANOTACIONES

Pasando parámetros a `raw()`:

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

Si necesita realizar consultas parametrizadas, puede usar el argumento `params` para `raw()`:

```
lname = 'Doe'
Person.objects.raw('SELECT * FROM myapp_person WHERE last_name = %s',
[lname])
```

Los parámetros no se pueden incluir entre comillas:

```
query ="SELECT * FROM myapp_person WHERE last_name = '%s'"
```

EJECUTANDO SQL PERSONALIZADO DIRECTAMENTE

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()` para obtener 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.

Por ejemplo:

```
from django.db import connection
def my_custom_sql(self):
    with connection.cursor() as cursor:
        cursor.execute("UPDATE bar SET foo = 1 WHERE baz = %s", [self.baz])
        cursor.execute("SELECT foo FROM bar WHERE baz = %s", [self.baz])
        row = cursor.fetchone()
return row
```

Para protegerse contra la inyección de SQL, no debe incluir comillas alrededor de los marcadores de posición `%s` en la cadena de SQL. Se debe tener en cuenta que si quieres incluir signos de porcentaje literal en la consulta, tienes que duplicarlos en caso de que estés pasando parámetros:

```
cursor.execute("SELECT foo FROM bar WHERE baz = '30%'")
cursor.execute("SELECT foo FROM bar WHERE baz = '30%%' AND id = %s",
[self.id])
```

Si está utilizando más de una base de datos, puede usar `django.db.connections` para obtener la conexión (y el cursor) para una base de datos específica. `django.db.connections` es un objeto similar a un diccionario, el cual permite recuperar una conexión específica utilizando su alias:

```
from django.db import connections
with connections['my_db_alias'].cursor() as cursor:
    # Tu código aquí …
```

De forma predeterminada, la API de Python DB devolverá resultados sin sus nombres de campo, lo que significa que terminará con una lista de valores, en lugar de un `dict`. Con un bajo costo de rendimiento y memoria, puede devolver los resultados como un `dict` usando algo como esto:

```
def dictfetchall(cursor):
    "Return all rows from a cursor as a dict"
    columns = [col[0] for col in cursor.description]
    return [
        dict(zip(columns, row))
        for row in cursor.fetchall()
    ]
```

Otra opción es usar `collections.namedtuple()` de la biblioteca estándar de Python. Un `namedtuple` es un objeto similar a una tupla que tiene campos accesibles por búsqueda de atributos. También es indexable e iterable. Los resultados son inmutables y accesibles por nombres de campo o índices, que pueden ser útiles:

```
from collections import namedtuple
def namedtuplefetchall(cursor):
    "Return all rows from a cursor as a namedtuple"
    desc = cursor.description
    nt_result = namedtuple('Result', [col[0] for col in desc])
    return [nt_result(*row) for row in cursor.fetchall()]
```

Aquí hay un ejemplo de la diferencia entre los tres:

```
>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");
>>> cursor.fetchall()
((54360982, None), (54360880, None))

>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");
>>> dictfetchall(cursor)
[{'parent_id': None, 'id': 54360982}, {'parent_id': None, 'id': 54360880}]
>>> cursor.execute("SELECT id, parent_id FROM test LIMIT 2");
>>> results = namedtuplefetchall(cursor)
>>> results
[Result(id=54360982, parent_id=None), Result(id=54360880, parent_id=None)]
>>> results[0].id
54360982
>>> results[0][0]
54360982
```

CONEXIONES Y CURSORES

`connection` y el cursor implementan principalmente la API Python DB-API estándar descrita en PEP 249, excepto cuando se trata del manejo de transacciones.

Si no está familiarizado con Python DB-API, tenga en cuenta que la instrucción SQL en `cursor.execute()` usa marcadores de posición, `"%s"`, en lugar de agregar parámetros directamente dentro del SQL. Si utiliza esta técnica, la biblioteca de base de datos subyacente escapará automáticamente de sus parámetros según sea necesario. También tenga en cuenta que Django espera el marcador de posición `"%s"`, no el `"?"` marcador de posición, que utilizan los enlaces SQLite Python. Esto es por el bien de la consistencia y la cordura.

Usando un cursor como gestor de contexto:

```
with connection.cursor() as c:
    c.execute(...)
es equivalente a:
c = connection.cursor()
try:
    c.execute(...)
finally:
    c.close()
```

INVOCACIÓN A PROCEDIMIENTOS ALMACENADOS

`CursorWrapper.callproc(procname, params=None, kparams=None)` llama a un procedimiento almacenado de base de datos con el nombre dado. Se puede proporcionar una secuencia (`params`), o diccionario (`kparams`) de parámetros de entrada. La mayoría de las bases de datos no son compatibles con `kparams`. De los backends integrados de Django, solo Oracle lo admite. Por ejemplo, dado este procedimiento almacenado en una base de datos Oracle:

```
CREATE PROCEDURE "TEST_PROCEDURE"(v_i INTEGER, v_text NVARCHAR2(10)) AS
    p_i INTEGER;
    p_text NVARCHAR2(10);
BEGIN
    p_i := v_i;
    p_text := v_text;
    ...
END;
```

Esto lo llamará:

```
with connection.cursor() as cursor:
    cursor.callproc('test_procedure', [1, 'test'])
```