TEXT CLASS REVIEW

TEMAS A TRATAR EN EL CUE:

- Manejo de relaciones en el ORM de Django
- Relaciones Muchos a Uno
- Relaciones Muchos a Muchos
- Relaciones Uno a Uno

¿QUÉ SON LAS RELACIONES EN EL ORM DE DJANGO?

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

Las relaciones en el ORM de Django tienen 3 tipos de campos para establecerlas entre los modelos (Tablas):

- El campo `ForeignKey`, que se utiliza para establecer una relación de 1 registro a varios registros.
- El campo `OneToOneField`, que se utiliza para relacionar registros uno a uno.
- El campo `ManyToManyField`, que se utiliza para relacionar varios registros entre sí, creando una nueva tabla que contiene los IDs de los modelos.

¿QUÉ ES UNA RELACIÓN MUCHOS A UNO?

Es el tipo de relación más común. Significa que un registro de la tabla A puede tener muchos registros coincidentes en la tabla B, pero un registro de la tabla B sólo tiene un registro coincidente en la tabla A.

¿CUÁNDO SE UTILIZAN LAS RELACIONES MUCHOS A UNO?

La relación se utiliza con frecuencia para describir clasificaciones o agrupaciones. Por ejemplo: en un esquema geográfico que tenga las tablas Región, Estado y Ciudad, muchos estados pertenecen a una región determinada, pero los mismos estados no pueden pertenecer a dos regiones diferentes. Lo mismo ocurre con las ciudades, una ciudad sólo está en un estado. Cada ciudad existe en un solo estado, pero un estado puede tener muchas ciudades, de ahí el término muchos a uno.

¿CÓMO SE IMPLEMENTA?

Representar la relación de uno a muchos utilizando un modelo de unión / intermedio. Veamos el siguiente código:

```
class Number(models.Model):
    number = models.CharField(max_length = 10)

class Person(models.Model):
    name = models.CharField(max_length = 200)

class PersonNumber(models.Model):
    person = models.ForeignKey(Person, on_delete = models.CASCADE, related_name = "numbers")
    number = models.ForeignKey(Number, on_delete = models.CASCADE, related_name = "person")
```

El modelo `Number` tiene un campo `number` para almacenar un número de teléfono. El modelo `Person` tiene un campo `name` para el nombre de la persona. `PersonNumber` es un modelo intermedio, o de unión, para `Number` y `Person`. Las claves foráneas tienen una relación en cascada con los objetos referidos. Este modelo se puede utilizar para establecer una relación de uno a varios, así como una relación de varios a varios.

La clave primaria predeterminada para todos los modelos es `id`, un campo automático de enteros. Dado que un número de teléfono está asociado con una sola persona, pero una persona puede tener más de un número de teléfono, esta es una relación de uno a varios. Se utilizará `PersonNumber` para representarla.

Si una variable `person` está almacenando un objeto `Person`, podemos acceder fácilmente a todos los números de teléfono de esta persona utilizando la siguiente declaración:

```
numbers = person.numbers.objects.all()
```

Esta declaración devolverá un `QuerySet` de objetos `Number`. Representar relaciones de una a varias mediante claves externas. Veamos el siguiente código:

```
numbers = person.numbers.objects.all()
```

El modelo `Person` tiene un campo `name` para el nombre de la persona. El modelo `Number` tiene un campo `number` para almacenar un número de teléfono, y una referencia de clave externa al modelo `Person`. Este

campo almacenaría al propietario de este número. La clave externa tiene una relación en cascada con el modelo referenciado, `Person`.

Usando esta estructura, podemos asociar fácilmente cada objeto `Number` con su propietario respectivo usando la referencia de clave externa. Si una variable `person` está almacenando un objeto `Person`, podemos acceder a todos los números asociados con esta persona utilizando la siguiente declaración:

```
numbers = Number.objects.filter(person = person)
```

Esta declaración devolverá un `QuerySet` de objetos `Number`.

DEFINICIÓN DE BORRADO EN CASCADA

Éste se aplica cuando creamos una clave foránea utilizando esta opción, se eliminan las filas de referencia en la tabla secundaria cuando la fila referenciada se elimina en la tabla primaria que tiene una clave primaria.

El método `on_delete` se utiliza para indicarle a Django qué hacer con las instancias de modelo que dependen de la instancia de modelo que elimine (por ejemplo, una relación `ForeignKey`). El `on_delete=models.CASCADE` le indica a Django que aplique el efecto de eliminación en cascada, es decir, que continúe eliminando también los modelos dependientes.

Aquí hay un ejemplo más concreto. Supongamos que tiene un modelo `Author` que es un `ForeignKey` en un modelo Book. Ahora, si elimina una instancia del modelo `Author`, Django no sabría qué hacer con las instancias del modelo `Book` que depende de dicha instancia. El método `on_delete` le indica qué hacer en ese caso. El ajuste `on_delete=models.CASCADE` indicará que ejecute el efecto de eliminación en cascada, es decir, que elimine todas las instancias del modelo `Book` que dependen de la instancia del modelo `Author` que eliminó.

¿QUÉ ES UNA RELACIÓN MUCHOS A MUCHOS?

Se produce cuando varios registros de una tabla se asocian a varios registros de otra tabla. Por ejemplo, existe una relación de muchos a muchos entre los clientes y los productos: los clientes pueden comprar varios productos, y los productos pueden ser comprados por muchos clientes.

¿CUÁNDO SE UTILIZA LA RELACIÓN MUCHOS A MUCHOS?

Por ejemplo, existe una relación de muchos a muchos entre los clientes y los productos: los clientes pueden comprar varios productos, y los productos pueden ser comprados por muchos clientes.

Otro ejemplo típico es aquella entre los estudiantes y las clases. Un estudiante puede matricularse en muchas clases, y una clase puede incluir muchos estudiantes.

¿CÓMO SE IMPLEMENTA?

Se puede dividir la relación de muchos a muchos en dos relaciones de uno a muchos, mediante el uso de una tercera tabla denominada tabla de unión. Cada registro de una tabla de unión incluye un campo de coincidencia que contiene el valor de las claves principales de las dos tablas que se unen (en la tabla de unión, estos campos de coincidencia son claves externas). Estos campos de clave externa se rellenan con datos, ya que los registros de la tabla de unión se crean desde cualquiera de las tablas que se unen.

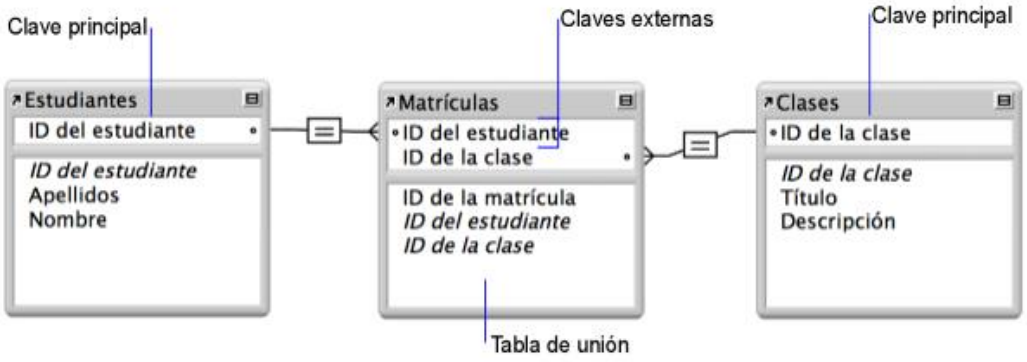
En Django, para establecer las relaciones Muchos a Muchos haciendo uso del campo `ManyToManyField` para relacionar varios registros entre sí, se crea una nueva tabla que contiene los `IDs` de los modelos.

Veamos un ejemplo. Un `Artículo` se puede publicar en múltiples `Publicaciones`, y una `Publicación` tiene múltiples `Artículos`:

```
from django.db import models
class Publicacion(models.Model):
    title = models.CharField(max_length=30)classMeta:
        ordering =['title']def__str__(self):return self.title
class Articulo(models.Model):
    headline = models.CharField(max_length=100)
    publicaciones = models.ManyToManyField(Publication)classMeta:
        ordering =['headline']def__str__(self):return self.headline
```

DEFINIENDO ENTIDADES INTERMEDIAS CON MÁS CAMPOS

En el ejemplo de los `estudiantes` y las `clases`, se incluye una tabla `Alumnos`, que contiene un registro para cada `estudiante`, y una tabla `Clases`, que contiene un registro para cada clase. Una tabla de unión o intermedia, `Matrículas`, crea una relación de uno a muchos, una entre cada una de las dos tablas.



Tablas `Alumnos` y `Clases`, cada una con una línea de relación con la tabla de unión `Matrículas`.

La clave principal `ID` de estudiante identifica de forma exclusiva a cada estudiante de la tabla `Alumnos`. La clave principal `ID` de clase identifica de forma exclusiva cada clase de la tabla `Clases`. La tabla `Matrículas` contiene las claves externas `ID de estudiante` e `ID de clase`.

Generalmente, las tablas de unión contienen campos que no tienen sentido en otras tablas. Puede añadir campos a la tabla `Matrículas`, como `Fecha` para mantener un registro de cuándo alguien inició una clase, y `Coste` para rastrear cuánto pagó un estudiante por realizar una clase. Mediante este diseño, si un estudiante se matricula en tres clases, tendrá un registro en la tabla `Alumnos`, y tres registros en la tabla `Matrículas`: un registro para cada clase en la que se ha matriculado. Las tablas de unión o intermedias pueden acceder a los campos y datos entre tablas sin necesidad de crear una relación diferente.

¿QUÉ ES UNA RELACIÓN UNO A UNO?

Es un vínculo entre la información de dos tablas, donde cada registro en cada tabla solo aparece una vez. En las relaciones de este tipo cada registro de la "Tabla padre" solo puede tener un registro enlazado en la "tabla hija", y cada registro de la "tabla hija" solo puede tener como máximo un registro enlazado con la "tabla padre".

¿CUÁNDO SE UTILIZA LA RELACIÓN UNO A UNO?

Este tipo de relación no es normal, pues la mayoría de la información que se relaciona de esta forma estaría en una tabla. Puede utilizarse la relación uno a uno para dividir una tabla con muchos campos, para aislar parte de una tabla por razones de seguridad, o para almacenar información que sólo se aplica a un subconjunto de la tabla principal.

Un ejemplo de este tipo de relaciones podría ser una tabla que relaciona cada trabajador con otra tabla en la que guardará su "Curriculum Vitae"; es decir, que cada trabajador solo tiene un currículo, y cada currículo solo tiene referencia a un trabajador.

¿CÓMO SE IMPLEMENTA?

Para definir una relación uno a uno, use `OneToOneField` como se muestra a continuación:

```
from django.db import models
class Publicacion(models.Model):
    title = models.CharField(max_length=30)classMeta:
        ordering =['title']def__str__(self):return self.title
class Articulo(models.Model):
    headline = models.CharField(max_length=100)
    publicaciones = models.ManyToManyField(Publication)classMeta:
        ordering =['headline']def__str__(self):return self.headline
```