Contenidos a Trabajar

1. Relaciones
   - Relaciones de muchos a uno
   - Relaciones de muchos a muchos
   - Relaciones de uno a uno
2. Herencia
   - Modelos proxy
3. Haciendo consultas

Relaciones

Claramente, el poder de las bases de datos relacionales radica en relacionar las tablas entre sí. Django ofrece formas de definir los tres tipos más comunes de relaciones de base de datos: muchos a uno, muchos a muchosy uno a uno.

Relaciones de muchos a uno

Para definir una relación de muchos a uno, utilice **django.db.models.ForeignKey**. Lo usa como cualquier otro tipo de **Field**: incluyéndolo como un atributo de clase de su modelo.

ForeignKey requiere un argumento posicional: la clase con la que se relaciona el modelo.

Por ejemplo, si un modelo **Car** tiene un **Manufacturer**, es decir, un **Manufacturer** fabrica varios autos pero cada **Car** solo tiene un **Manufacturer**, use las siguientes definiciones:

```
from django.db import models

class Manufacturer(models.Model):
    # ...
    pass

class Car(models.Model):
    manufacturer = models.ForeignKey(Manufacturer, on_delete=models.CASCADE)
    # ...
```

También puede crear relaciones recursivas (un objeto con una relación de varios a uno consigo mismo) y relaciones con modelos aún no definidos; vea la documentación oficial para más detalles.

Se sugiere, pero no es obligatorio, que el nombre de un campo **ForeignKey** (**manufacturer** en el ejemplo anterior) sea el nombre del modelo, en minúsculas. Puedes llamar al campo como quieras.

Relaciones de muchos a muchos

Para definir una relación de muchos a muchos, utilice **ManyToManyField**. Lo usa como cualquier otro tipo de **Field**: incluyéndolo como un atributo de clase de su modelo.

ManyToManyField requiere un argumento posicional: la clase con la que se relaciona el modelo.

Por ejemplo, si **Pizza** tiene varios objetos **Topping**, es decir, **Topping** puede estar en varias pizzas y cada **Pizza** tiene varios ingredientes (toppings), así es como lo representarías:

```
from django.db import models

class Topping(models.Model):
# ...
pass

class Pizza(models.Model):
# ...
toppings = models.ManyToManyField(Topping)
```

Al igual que con **ForeignKey**, también puede crear relaciones recursivas (un objeto con una relación de muchos a muchos consigo mismo) y relaciones con modelos aún no definidos.

Se sugiere, pero no se requiere, que el nombre del **ManyToManyField** (**toppings** en el ejemplo anterior) sea un plural que describa el conjunto de objetos de modelo relacionados.

No importa qué modelo tenga el **ManyToManyField**, pero solo debe colocarlo en uno de los modelos, no en ambos.

Generalmente, las instancias **ManyToManyField** deben ir en el objeto que se va a editar en un formulario. En el ejemplo anterior, **toppings** está en **Pizza** (en lugar de **Topping** tener un **pizzas ManyToManyField**) porque es más natural pensar en una pizza con ingredientes que en varias pizzas. De la forma en que está configurado arriba, el formulario **Pizza** permitiría a los usuarios seleccionar los ingredientes.

Relaciones uno a uno

Para definir una relación uno a uno, utilice **OneToOneField**. Lo usa como cualquier otro tipo de **Field**:incluyéndolo como un atributo de clase de su modelo.

Esto es más útil en la clave principal de un objeto cuando ese objeto "extiende" otro objeto de alguna manera.

OneToOneField requiere un argumento posicional: la clase con la que se relaciona el modelo.

Por ejemplo, si estuviera creando una base de datos de «lugares», crearía cosas bastante estándar como direcciones, números de teléfono, etc. en la base de datos. Entonces, si quisieras construir una base de datos de restaurantes encima de los lugares, en lugar de repetirte y replicar esos campos en el modelo **Restaurant**, podrías hacer que **Restaurant** tenga un **OneToOneField** a **Place** (porque un restaurante «es un» lugar; de hecho, para manejar esto normalmente usaría la **herencia** (próximo doc), que implica una relación implícita deuno a uno).

Al igual que con **ForeignKey**, se puede definir una relación recursiva y se pueden hacer referencias a modelosaún no definidos.

Para más información sobre los campos y tipos de atributos que pueden tener, visite la documentación oficial:

https://docs.djangoproject.com/es/3.2/topics/db/models/?lang=es

Herencia

La herencia de modelos en Django funciona de manera casi idéntica a la forma en que funciona la herencia de clases normal en Python, pero aún se deben seguir los conceptos básicos de modelos. Eso significa que la clase base debería subclasificarse **django.db.models.Model**.

La única decisión que debe tomar es si desea que los modelos principales sean modelos por derecho propio (con sus propias tablas de base de datos), o si los principales son solo titulares de información común que solo será visible a través de los modelos secundarios.

Hay tres estilos de herencia posibles en Django.

1. A menudo, solo querrá usar la clase principal para contener información que no desea tener que escribir para cada modelo secundario. Esta clase nunca se utilizará de forma aislada, por lo que lo que busca son las **clases base abstractas**.
2. Si está subclasificando un modelo existente (quizás algo de otra aplicación por completo) y desea que cada modelo tenga su propia tabla de base de datos, la **herencia de tablas múltiples** es el camino a seguir.
3. Finalmente, si solo desea modificar el comportamiento a nivel de Python de un modelo, sin cambiar los campos de los modelos de ninguna manera, puede usar **modelos Proxy**.

Clases base abstractas

Las clases base abstractas son útiles cuando desea poner información común en una serie de otros modelos. Escribes tu clase base y la pones **abstract=True** en la clase Meta. Este modelo no se utilizará para crear ninguna tabla de base de datos. En cambio, cuando se usa como clase base para otros modelos, sus campos se agregarán a los de la clase secundaria.

Un ejemplo:

```
from django.db import models
class CommonInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()
    class Meta:
```

```
        abstract = True
class Student(CommonInfo):
    home_group = models.CharField(max_length=5)
```

El modelo **Student** tendrá tres campos: **name**, **age** y **home_group**. El modelo **CommonInfo** no se puede usar como un modelo normal de Django, ya que es una clase base abstracta. No genera una tabla de base de datos ni tiene un administrador, y no se puede instanciar o guardar directamente.

Los campos heredados de clases base abstractas se pueden anular con otro campo o valor, o eliminarsecon **None**.

Para muchos usos, este tipo de herencia de modelo será exactamente lo que desea. Proporciona una forma de factorizar la información común en el nivel de Python, al mismo tiempo que solo crea una tabla de base de datos por modelo secundario en el nivel de la base de datos.

Herencia de tablas múltiples

El segundo tipo de herencia de modelo soportado por Django es cuando cada modelo en la jerarquía es un modelo en sí mismo. Cada modelo corresponde a su propia tabla de base de datos y se puede consultar y crear individualmente. La relación de herencia introduce vínculos entre el modelo hijo y cada uno de sus padres (a través de un campo **OneToOneField**). Por ejemplo:

```
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)
```

Todos los campos de **Place** también estarán disponibles en **Restaurant**, aunque los datos residirán en una tabla de base de datos diferente. Así que ambos son posibles.

Si tiene un **Place** que también es un **Restaurant**, puede pasar del objeto **Place** al objeto **Restaurant** usando laversión en minúsculas del nombre del modelo:

```
>>> p = Place.objects.get(id=12)
# If p is a Restaurant object, this will give the child class:
>>> p.restaurant
<Restaurant: ...>
```

Modelos proxy

Cuando se utiliza la **herencia de varias tablas**, se crea una nueva tabla de base de datos para cada subclase de un modelo. Este suele ser el comportamiento deseado, ya que la subclase necesita un lugar para almacenar cualquier campo de datos adicional que no esté presente en la clase base. A veces, sin embargo, solo desea cambiar el comportamiento de Python de un modelo, tal vez para cambiar el administrador predeterminado oagregar un nuevo método.

Para esto sirve la herencia del modelo proxy: crear un *proxy* para el modelo original. Puede crear, eliminar y actualizar instancias del modelo proxy y todos los datos se guardarán como si estuviera usando el modelo original (sin proxy). La diferencia es que puede cambiar cosas como el orden predeterminado del modelo o el administrador predeterminado en el proxy, sin tener que alterar el original.

Los modelos proxy se declaran como modelos normales. Le dices a Django que es un modelo de proxyconfigurando el atributo **proxy** de la clase **Meta** en **True**.

Por ejemplo, suponga que desea agregar un método al modelo **Person**. Puedes hacerlo así:

```
from django.db import models
class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
class MyPerson(Person):
    class Meta:
        proxy = True
    def do_something(self):
    # ...
    Pass
```

Para más información sobre el manejo de herencia y las características particulares de cada caso, así como el manejo de herencia múltiple, por favor visite la documentación oficial en:

https://docs.djangoproject.com/es/3.2/topics/db/models/

Haciendo consultas

Una vez que haya creado sus modelos de datos , Django le brinda automáticamente una API de abstracción de base de datos que le permite crear, recuperar, actualizar y eliminar objetos. Este documento explica cómo utilizar esta API. Consulte la referencia del modelo de datos para obtener detalles completos de todas las diversas opciones de búsqueda de modelos.

A lo largo de esta guía, nos referiremos a los siguientes modelos, que comprenden una aplicación Weblog:

```
from django.db import models

class Blog(models.Model):
        name = models.CharField(max_length=100)
        tagline = models.TextField()
        def __str__(self):
                return self.name

class Author(models.Model):
        name = models.CharField(max_length=200)
        email = models.EmailField()
        def __str__(self):
                return self.name

class Entry(models.Model):
        blog = models.ForeignKey(Blog, on_delete=models.CASCADE)
        headline = models.CharField(max_length=255)
        body_text = models.TextField()
        pub_date = models.DateField()
        mod_date = models.DateField()
        authors = models.ManyToManyField(Author)
        number_of_comments = models.IntegerField()
        number_of_pingbacks = models.IntegerField()
        rating = models.IntegerField()

        def __str__(self):
                return self.headline
```

Creando objetos

Para representar los datos de la tabla de la base de datos en los objetos de Python, Django utiliza un sistema intuitivo: una clase modelo representa una tabla de la base de datos y una instancia de esa clase representa un registro particular en la tabla de la base de datos.

Para crear un objeto, cree una instancia usando argumentos de palabras clave para la clase modelo, luegollame **save()** para guardarlo en la base de datos.

Asumiendo que los modelos viven en un archivo **mysite/blog/models.py**, aquí hay un ejemplo:

```
>>> from blog.models import Blog
>>> b = Blog(name='Beatles Blog', tagline='All the latest Beatles news.')
>>> b.save()
```

Esto realiza una instrucción **INSERT** SQL detrás de escena. Django no llega a la base de datos hasta que llamas explícitamente a **save()**. El método **save()** no tiene valor de retorno.

Guardar cambios en objetos

Para guardar cambios en un objeto que ya está en la base de datos, use **save()**.

Dada una instancia **Blog b5** que ya se ha guardado en la base de datos, este ejemplo cambia su nombre yactualiza su registro en la base de datos:

```
>>> b5.name = 'New name'
>>> b5.save()
```

Esto realiza una instrucción **UPDATE** SQL detrás de escena. Django no llega a la base de datos hasta que llamasexplícitamente a **save()**.

Recuperando objetos

Para recuperar objetos de su base de datos, construya un **QuerySet** vía a **Manager** en su clase de modelo.

Un **QuerySet** representa una colección de objetos de su base de datos. Puede tener cero, uno o varios *filtros*. Los filtros reducen los resultados de la consulta en función de los parámetros dados. En términos de SQL, a **QuerySet** equivale a una declaración **SELECT** y un filtro es una cláusula limitante como **WHERE** o **LIMIT**.

Obtienes un **QuerySet** mediante el uso del **Manager** del modelo. Cada modelo tiene al menos un **Manager**, y se llama **objects** de forma predeterminada. Acceda directamente a través de la clase modelo, así:

```
>>> Blog.objects
<django.db.models.manager.Manager object at ...>
>>> b = Blog(name='Foo', tagline='Bar')
>>> b.objectsTraceback:
...
AttributeError: "Manager isn't accessible via Blog instances."
```

Los **Managers** son accesibles solo a través de clases de modelo, en lugar de instancias de modelo, para imponer una separación entre las operaciones de "nivel de tabla" y las operaciones de "nivel de registro".

El **Manager** es la fuente principal de **QuerySets** para un modelo. Por ejemplo, **Blog.objects.all()** devuelve un **QuerySet** que contiene todos los objetos **Blog** de la base de datos.

Recuperando objetos específicos con filtros

El **QuerySet** devuelto por **all()** describe todos los objetos en la tabla de la base de datos. Sin embargo, por lo general, deberá seleccionar solo un subconjunto del conjunto completo de objetos.

Para crear dicho subconjunto, refina el **QuerySet** inicial y agrega condiciones de filtro. Las dos formas más comunes derefinar a **QuerySet** son:

filter(**kwargs)

Devuelve un nuevo objeto **QuerySet** contenedor que coincide con los parámetros de búsqueda dados.

exclude(kwargs)**

Devuelve un nuevo objeto **QuerySet** que contiene objetos que *no* coinciden con los parámetros de búsqueda proporcionados.

Los parámetros de búsqueda (**kwargs) en las definiciones de funciones anteriores deben tener el formato descrito en Búsquedas de campos a continuación.

Por ejemplo, para obtener una **QuerySet** de las entradas de blog del año 2006, use **filter()** así:

```
Entry.objects.filter(pub_date__year=2006)
```

Con la clase de manager predeterminada, es lo mismo que:

```
Entry.objects.all().filter(pub_date__year=2006)
```

Los QuerySet son Lazy

Los **QuerySets** son perezosos (lazy): el acto de crear **QuerySet** no implica ninguna actividad de base de datos. Puede apilar filtros durante todo el día, y Django no ejecutará la consulta hasta **QuerySet** que se *evalúe*. Echale un vistazo a éste ejemplo:

```
>>> q = Entry.objects.filter(headline__startswith="What")
>>> q = q.filter(pub_date__lte=datetime.date.today())
>>> q = q.exclude(body_text__icontains="food")
>>> print(q)
```

Aunque esto parece tres visitas a la base de datos, de hecho, llega a la base de datos solo una vez, en la última línea (**print(q)**). En general, los resultados de un **QuerySet** no se recuperan de la base de datos hasta que usted los "solicita". Cuando lo hace, **QuerySet** se *evalúa* accediendo a la base de datos. Para obtener más detalles sobre cuándo se lleva a cabo exactamente la evaluación, consulte la documentación oficial.

Para más información sobre filtros, distintas formas de realizar queries y la especificación de la api, vaya a la documentación oficial:

https://docs.djangoproject.com/es/3.2/topics/db/queries/