Tidbits | Aug. 31, 2018

Tips for Using Django's ManyToManyField

by Lacey Williams Henschel | More posts by Lacey

Versions:

- Python 3.7
- Django 2.1

`ManyToManyField`s confuse a lot of people. The way you relate objects to each other using a many-to-many relationship is just different enough from dealing with `ForeignKey`s and just uncommon enough in day-to-day Django development that it's easy to forget all the little tricks for dealing with them.

When should you use a `ManyToManyField` instead of a regular `ForeignKey`? To remember that, let's think about pizza. A pizza can have many toppings (a Hawaiian pizza usually has Canadian bacon and pineapple), and a topping can go on many pizzas (Canadian bacon also appears on meat lovers' pizzas). Since a pizza can have more than one topping, and a topping can go on more than one pizza, this is a great place to use a `ManyToManyField`.

So let's dive in: assume the following models in a `pizzas` app.

```
 django.db import models


; Pizza(models.Model):

 ame = models.CharField(max_length=30)
:oppings = models.ManyToManyField('Topping')

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

```
name = models.CharField(max_length=30)

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

Both objects must exist in the database

You have to save a `Topping` in the database before you can add it to a `Pizza`, and vice versa. This is because a `ManyToManyField` creates an invisible "through" model that relates the source model (in this case `Pizza`, which contains the `ManyToManyField`) to the target model (`Topping`). In order to create the connection between a pizza and a topping, they both have to be added to this invisible "through" table. From the Django docs:

" [T]here is … an implicit through model class you can use to directly access the table created to hold the association. It has three fields to link the models. If the source and target models differ, the following fields are generated:

- `id`: the primary key of the relation.
- `<containing_model>_id`: the id of the model that declares the ManyToManyField.
- `<other_model>_id`: the id of the model that the ManyToManyField points to."

The invisible "through" model that Django uses to make many-to-many relationships work requires the primary keys for the source model and the target model. A primary key doesn't exist until a model instance is saved, so that's why both instances have to exist before they can be related. (You can't add spinach to your pizza if you haven't bought spinach yet, and you can't add spinach to your pizza if you haven't even started rolling out the crust yet either.)

See what happens when you try to add a topping to a pizza before you've added that topping to the database:

```
from pizzas.models import Pizza, Topping
hawaiian_pizza = Pizza.objects.create(name='Hawaiian')
pineapple = Topping(name='pineapple')
hawaiian_pizza.toppings.add(pineapple)
Traceback (most recent call last):

ValueError: Cannot add "<Topping: pineapple>": instance is on database "default",
e is on database "None"
```

A `ValueError` is raised because the `pineapple` hasn't yet been saved, so its value on the database doesn't exist yet. But when I save `pineapple`, I can add it to my pizza.

```
ySet [<Topping: pineapple>]>
```

The reverse doesn't work either: I can't create a topping in the database, and then add it to a pizza that hasn't been saved.

```
epperoni = Topping.objects.create(name='pepperoni')
epperoni_pizza = Pizza(name='Pepperoni')
epperoni_pizza.toppings.add(pepperoni)
eback (most recent call last):

eError: "<Pizza: Pepperoni>" needs to have a value for field "id" before this many
tionship can be used.
```

This error is more explicit (it states that an id is required) but it's essentially the same error. It's just coming from the other side of the relationship.

To retrieve the stuff in a `ManyToManyField`, you have to use `*_set` ...

Since the field `toppings` is already on the `Pizza` model, getting all the toppings on a specific pizza is pretty straightforward.

```
awaiian_pizza.toppings.all()
ySet [<Topping: pineapple>, <Topping: Canadian bacon>]>
```

But if I try to see what pizzas use Canadian bacon, I get an `AttributeError`:

```
anadian_bacon.pizzas.all()
eback (most recent call last):

tbuteError: 'Topping' object has no attribute 'pizzas'
```

That's because Django automatically refers to the target `ManyToManyField` objects as "sets." The pizzas that use specific toppings are in their own "set":

```
anadian_bacon.pizza_set.all()
ySet [<Pizza: Hawaiian>]>
```

... unless you add the `related_name` option to the field

```
; Pizza(models.Model):
...
:oppings = models.ManyToManyField('Topping', related_name='pizzas')
```

The `related_name` should usually be the lowercase, plural form of your model name. This is confusing for some people because shouldn't the `related_name` for `toppings` just be… toppings?

No; the `related_name` isn't referring to how you want to retrieve the stuff *in this field*; it specifies the term you want to use instead of `*_set` when you're on the target object (which in this case is a topping) and want to see which source objects point to that target (what pizzas use a specific topping).

Without a `related_name`, we would retrieve all the pizzas that use a specific topping with `pizza_set`:

```
anadian_bacon.pizza_set.all()
ySet [<Pizza: Hawaiian>]>
```

Adding a `related_name` of "pizzas" to the `toppings` attribute lets us retrieve all the toppings for a pizza like this:

```
anadian_bacon.pizzas.all()
ySet [<Pizza: Hawaiian>]>
```

You can add things from both sides of the relationship.

Earlier, we created a `Pizza` object, and then a `Topping` object, and then ran `hawaiian_pizza.toppings.add(pineapple)` to associate the pineapple topping with the Hawaiian pizza.

But we could, instead, add a pizza to a topping.

```
eese_pizza = Pizza.objects.create(name='Cheese')
ozzarella = Topping.objects.create(name='mozzarella')
ozzarella.pizzas.add(cheese_pizza)
ozzarella.pizzas.all()
ySet [<Pizza: Cheese>]>
```

You can query the items in the many-to-many set from both sides

Say we want to find all the pizzas that have toppings that start with the letter "p." We can write that query like this:

A Hawaiian pizza contains pineapple, and a pepperoni pizza contains pepperonis. Pineapple and pepperoni both start with the letter "p," so the both of those pizzas are returned.

We can do the same from the `Topping` model, to find all the toppings used on pizzas that contain "Hawaiian" in their name:

```
>pping.objects.filter(pizzas__name__contains='Hawaiian')
-ySet [<Topping: pineapple>, <Topping: Canadian bacon>]>
```

You might need a custom "through" model

Remember when I mentioned the invisible "through" model that Django creates to manage your many-to-many relationships? You might want to keep track of more data about those relationships, and to do that you would use a custom "through" model.

The example used in the Django docs is of a Group, Person, and Membership relationship. A group can have many people as members, and a person can be part of many groups, so the `Group` model has a `ManyToManyField` that points to `Person`. Then, a `Membership` model contains `ForeignKey`s to both `Person` and `Group`, and can store extra information about a person's membership in a specific group, like the date they joined, who invited them, etc.

But we're not here to talk about people. We are all about pizza.

Using our existing models, we can create all kinds of pizzas with a wide range of toppings. But we can't make a pizza like "Super Pepperoni" that contains double the usual amount of pepperonis. We can't add pepperoni to a pizza more than once:

```
epperoni_pizza.toppings.all()
-ySet [<Topping: pepperoni>]>
epperoni_pizza.toppings.add(pepperoni)
epperoni_pizza.toppings.all()
-ySet [<Topping: pepperoni>]>
```

Django just ignores us if we try. A "through" model would let us specify a quantity for each topping, enabling us to add "pepperoni" once, but specify that we wanted twice the amount for the Super Pepperoni pizza.

Note: If you're going to use a "through" model, you have to start with that in mind… or be willing to either drop your database or do some very advanced database finagling. If you try to add a "through" model later, you will see an error like this one when you run `migrate`:

```
eError: Cannot alter field pizzas.Pizza.toppings into pizzas.Pizza.toppings -
 are not compatible types (you cannot alter to or from M2M fields, or add or remo
```

```
; ToppingAmount(models.Model):

REGULAR = 1
OUBLE = 2
RIPLE = 3
MOUNT_CHOICES = (
    (REGULAR, 'Regular'),
    (DOUBLE, 'Double'),
    (TRIPLE, 'Triple'),
)

izza = models.ForeignKey('Pizza', related_name='topping_amounts', on_delete=mode
opping = models.ForeignKey('Topping', related_name='topping_amounts', on_delete=
mount = models.IntegerField(choices=AMOUNT_CHOICES, default=REGULAR)
```

Now, add the `through` option to the `toppings` field on the `Pizza` model:

```
; Pizza(models.Model):
.
oppings = models.ManyToManyField('Topping', through='ToppingAmount', related_nam
```

And run `makemigrations` and `migrate`.

If we try to add a pizza, a topping, and then associate them the way we used to, we will get an error:

```
per_pep = Pizza.objects.create(name='Super Pepperoni')
epperoni = Topping.objects.create(name='pepperoni')
per_pep.toppings.add(pepperoni)
eback (most recent call last):

buteError: Cannot use add() on a ManyToManyField which specifies an intermediary
izzas.ToppingAmount's Manager instead.
```

Using a custom "through" model forces us to use that model to associate the pizza and toppings.

```
mount.objects.create(pizza=super_pep, topping=pepperoni, amount=ToppingAmount.DO
```

But the benefit is that we can now add some extra information about that relationship, like the fact that the amount of pepperonis on a Super Pepperoni pizza should be double the regular

```
uper_pep.toppings.all()
ySet [<Topping: pepperoni>]>
```

We'll only see pepperoni once, since the amount is on the "through" model. And we can access the pizzas that use a specific topping:

```
epperoni.pizzas.all()
ySet [<Pizza: Super Pepperoni>]>
```

But now, we can use our "through" model to get all the toppings and their amount for a specific pizza from the `ToppingAmount` model:

```
top_amt in ToppingAmount.objects.filter(pizza=super_pep):
print(top_amt.topping.name, top_amt.get_amount_display())

eroni Double
```

You can also see the topping amounts from the `Pizza` objects themselves.

```
top_amt in super_pep.topping_amounts.all():
print(top_amt.topping.name, top_amt.get_amount_display())

eroni Double
```

And you can also access the amount for a specific topping from the `topping` on the `Pizza` object.

```
topping in super_pep.toppings.all():
for top_amt in topping.topping_amount.all():
    print(topping.name, top_amt.amount, top_amt.get_amount_display())

eroni 2 Double
```

You could extend this `ToppingAmount` through model to hold information about the left and right halves of the pizza, or notes about topping preparation ("put peppers under cheese"). You can also add methods to the through model or the source/target models to more easily access some of the topping amount information.

A through model is also useful for relationships between players and teams; the through model could contain information about the players' positions, jersey numbers, and dates they joined the team. A through model joining movie theatres and films could contain the number of screens the film is showing on and the start and end run dates. Students' relationships to their

Further Reading

- The many-to-many relationships section of the Django docs contains a lot of great examples of using `ManyToManyField`.
- The explanation of `ManyToManyField` in the model field reference.
- Jacob Kaplan-Moss also has a great set of examples of using a many-to-many relationship with a custom "through" model.

Thank you to Monique Murphy and Jeff Triplett for their assistance.

---

django manytomanyfield through model