# Django Admin Interface Does Not Use Subclass's __unicode__()

Asked 16 years, 1 month ago Modified 11 years, 1 month ago Viewed 5k times

5

(Django 1.x, Python 2.6.x)

I have models to the tune of:

```
class Animal(models.Model):
  pass

class Cat(Animal):
  def __unicode__(self):
    return "This is a cat"

class Dog(Animal):
  def __unicode__(self):
    return "This is a dog"

class AnimalHome(models.Model):
  animal = models.ForeignKey(Animal)
```

I have instantiated no Animals, because this is supposed to be a virtual class. I have instantiated Cats and Dogs, but in the Admin Page for AnimalHome, my choices for Animals are displayed as "Animal object" (the default __unicode__(), I guess) as opposed to the __unicode__ I have defined for the two subclasses. Help.

---

The abstract base class issue is a red herring wrt to this question, I think. Even if Animal was not supposed to be abstract, I still have the problem that, for some reason, since the ForeignKey is defined on Animal and not one of its subclasses, the superclass method is being called instead of the subclass. In OO programming when you call object.method() you're supposed to get the lowest-subclass's implementation, and you have to do extra work to get any superclass's implementation. So why is it that having __unicode__ defined on the subclasses is not sufficient for --- actually the problem might be that __unicode__ is not getting called at all because introspection on the Animal class reveals that it's not defined. So maybe if I define __unicode__ for Animal and have it call subclasses' __unicode__ I could get the desired effect.

---

Okay, I think that I understand the ORM issues. Both these answers have helped me understand this, thank you. While experimenting with this, I discovered that when Django saves a subclassed model, it does two things: (1) it creates a row for the subclassed object in the superclass's table, and (2) it makes the PK in the subclass table identical to the PK assigned in the superclass table. This PK in the subclass table is named superclass_ptr. Based upon this I've concocted the following. I'd appreciate feedback.

```
Class Animal(models.Model)
  def __unicode__(self):
    if Dog.objects.filter(pk=self.pk).count() > 0:
      return unicode(Dog.objects.get(pk=self.pk))
    elif Cat.objects.filter(pk=self.pk).count() > 0:
      return unicode(Cat.objects.get(pk=self.pk))
    else:
      return "An Animal!"
```

---

It seems that Lawrence is most on-point wrt this question. Cat and Dog will have disjoint PK sets (and any subclass of Animal will have a PK identical to the record of its superclass), but unfortunately Django does not perform any work behind the scenes a la: "I'm an Animal. I know Animals have subclasses Dog and Cat. Specifically, I'm Animal number 3, and furthermore I just checked and there's a Cat number 3 too. That means that I'm actually Cat number 3". Even though this seems entirely possible and very reasonable (since a Cat won't do anything an Animal couldn't do itself) using Python's introspection. Thank you all.

python django django-admin

Share

Improve this question

Follow

edited Nov 26, 2008 at 1:50

community wiki
8 revs
Carl G

I don't think you should code a Django model this way. Try to define your data model in such a way that you can use one table (i.e. one model) for all your animals. Don't put the descision into you code. If you pickle you objects into a data base row, then you get something likt an object store. – Ber Nov 24, 2008 at 7:44

## 6 Answers

Sorted by: Highest score (default)

6

ForeignKey(Animal) is just that, a foreign key reference to a row in the Animal table. There's nothing in the underlying SQL schema that indicates that the table is being used as a superclass, so you get back an Animal object.

To work around this:

First, you want the base class to be non-abstract. This is necessary for the ForeignKey anyway, and also ensures that Dog and Cat will have disjunct primary key sets.

Now, Django implements inheritance using a OneToOneField. Because of this, **an instance of a base class that has a subclass instance gets a reference to that instance, named appropriately.** This means you can do:

```
class Animal(models.Model):
    def __unicode__(self):
        if hasattr(self, 'dog'):
            return self.dog.__unicode__()
        elif hasattr(self, 'cat'):
            return self.cat.__unicode__()
        else:
            return 'Animal'
```

This also answers your question to Ber about a **unicode**() that's dependent on other subclass attributes. You're actually calling the appropriate method on the subclass instance now.

Now, this does suggest that, since Django's already looking for subclass instances behind the scenes, the code could just go all the way and return a Cat or Dog instance instead of an Animal. You'll have to take up that question with the devs. :)

Share
Improve this answer
Follow

edited Nov 25, 2008 at 16:13

answered Nov 24, 2008 at 16:40
Lawrence
10.7k ● 5 ● 41 ● 52

---

6

You want an [Abstract base class](#) ("virtual" doesn't mean anything in Python.)

From the documentation:

```
class CommonInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True
```

---

Edit

"In OO programming when you call object.method() you're supposed to get the lowest-subclass's implementation."

True. But not the whole story.

This is not a OO issue. Or even a Python or Django issue. This is an ORM issue.

The question is "What object is reconstructed at the end of the FK reference?" And the answer is that there's no standard, obvious answer of how to handle the transformation from FK value to object.

I've got a row in `AnimalHome` with an `animals` value of 42. It refers to `Animal.objects.get(pk=42)`. Which subclass of Animal? Cat? Dog? How does the ORM layer know if it should do `Dog.objects.get(pk=42)` or `Cat.objects.get(pk=42)`?

"But wait," you say. "It should fetch the Animal object, not a Dog or Cat object." You can hope for that, but that's not how the Django ORM works. Each class is a distinct table. Cat and Dog are -- by definition -- separate tables, with separate queries. You're not using an object store. You're using ORM to relational tables.

---

Edit

First, your query only works if Dog and Cat share a common key generator, and don't have an overlapping set of PK's.

If you have a Dog with PK of 42 AND a Cat with PK of 42, you've got a problem. And since you can't easily control the key generation, your solution can't work.

Run Time Type Identification is bad. It's not Object-Oriented in a number of ways. Almost anything you can do to avoid RTTI is better than an ever-expanding sequence of if-statements to distinguish subclasses.

However, the model you're trying to build is -- specifically -- a pathological problem for ORM systems. Indeed, so specifically pathological that I'm almost willing to bet it's homework. [There are pathological problems for pure SQL systems, also. They often show up as homework.]

The issue is that the ORM cannot do what you think it should do. So you have two choices.

- Stop using Django.
- Do something Django does directly.
- Break OO design guidelines and resort to brittle things like RTTI, which make it remarkably hard to add another subclass of animals.

Consider this way to do RTTI -- it includes the class name as well as the PK

```
KIND_CHOICES = (
   ( "DOG", "Dog" ),
   ( "CAT", "Cat" ),
)

class Animal( models.Model ):
    kind = models.CharField( max_length= 1, choices=KIND_CHOICES )
    fk = models.IntegerField()
    def get_kind( self ):
        if kind == "DOG":
            return Dog.objects.get( pk = fk )
        elif kind == "CAT":
            return Cat.objects.get( pk = fk )
```

Share  
Improve this answer  
Follow

edited Nov 24, 2008 at 10:57

answered Nov 23, 2008 at 23:23  
S.Lott  
**391k** ● 82 ● 517 ● 788

but this doesn't fix the problem. Now Django offers me the (more helpful) error meessage that an abstract base class cannot be used as a foreign key. – Carl G Nov 24, 2008 at 0:31

That's pretty much that. Abstract base classes can't be used as a foreign key. – S.Lott Nov 24, 2008 at 0:51

I'd appreciate your input on my edit above adding the new definition of unicode to Animal. It relies upon Django not treating superclass/subclass tables as entirely separate; it stores the superclass id in the subclass as a pointer. – Carl G Nov 24, 2008 at 5:08

3

Django (and relational data bases in general) do not work this way. Even when using an ORM like Django's you don't work with class hierarchies like this.

There are two possible solutions to your problem:

(1) give a "name" attibute the the Animal model, then add entities with names from ['Dog', 'Cat']. This will show the animals' names in the foreign key selection box.

(2) If you **really need to** link your foreign key to different models (which really is not the usual way to use an RDBMS) you should read about Generic Relations in the docs on the contenttypes framework.

My advice is (1), though.

Share

Improve this answer

Follow

edited Nov 24, 2008 at 9:02

answered Nov 23, 2008 at 23:22

Ber

41.7k ● 16 ● 77 ● 89

If I use your first suggestion, how do I overcome the fact that Django is not using the __unicode__() of the subclass? Say Dog had the property is_german_shepherd and Dog's __unicode__() returned: "A Dog: (is a german shepherd?: %s )" % ("Yes" if is_german_shepherd else "No") – Carl G Nov 24, 2008 at 0:28

2

This is along the lines of what S.Lott suggested, but without the if/elif/..., which can become increasingly awkward and hard to maintain as the number of subclasses you need to support grows.

```
class Cat(models.Model):
    def __unicode__(self):
        return u'A Cat!'

class Dog(models.Model):
    def __unicode__(self):
        return u'A Dog!'

class Eel(models.Model):
    def __unicode__(self):
        return u'An Eel!'

ANIMALS = {
    'CAT': {'model': Cat, 'name': 'Cat'},
    'DOG': {'model': Dog, 'name': 'Dog'},
    'EEL': {'model': Eel, 'name': 'Eel'},
}
KIND_CHOICES = tuple((key, ANIMALS[key]['name']) for key in ANIMALS)

class Animal(models.Model):
    kind = models.CharField(max_length=3, choices=KIND_CHOICES)
    fk = models.IntegerField()
    def get_kind(self):
        return ANIMALS[self.kind]['model'].objects.get(pk=self.fk)
    def __unicode__(self):
        return unicode(self.get_kind())
```

Something very similar can also be done with Django's multi-table inheritance (search Django's docs for it). For example:

```
ANIMALS = {
    'CAT': {'model_name': 'Cat', 'name': 'Cat'},
    'DOG': {'model_name': 'Dog', 'name': 'Dog'},
    'EEL': {'model_name': 'Eel', 'name': 'Eel'},
}
KIND_CHOICES = tuple((key, ANIMALS[key]['name']) for key in ANIMALS)

class Animal(models.Model):
    kind = models.CharField(max_length=3, choices=KIND_CHOICES)
    def get_kind(self):
        return getattr(self, ANIMALS[self.kind]['model_name'].lower())
    def __unicode__(self):
        return unicode(self.get_kind())

class Cat(Animal):
    def __unicode__(self):
        return u'A Cat!'

class Dog(Animal):
    def __unicode__(self):
        return u'A Dog!'

class Eel(Animal):
    def __unicode__(self):
        return u'An Eel!'
```

I personally prefer the second option, since the subclasses' instances will have all of the fields defined in the parent class auto-magically, which allows for clearer and more concise code. (For instace, if the Animal class had a 'gender' field, then Cat.objects.filter(gender='MALE') would work).

Share

Improve this answer

Follow

edited Nov 24, 2008 at 14:22

community wiki
4 revs
taleinat

1

Regarding Generic Relations, note that normal Django queries cannot span GenerecForeignKey relations. Using multi-table inheritance avoids this issue at the cost of being a less generic solution.

From the docs:

> Due to the way GenericForeignKey is implemented, you cannot use such fields directly with filters (filter() and exclude(), for example) via the database API. They aren't normal field objects. These examples will not work:

```
# This will fail
>>> TaggedItem.objects.filter(content_object=guido)
# This will also fail
>>> TaggedItem.objects.get(content_object=guido)
```

Share
Improve this answer
Follow

answered Nov 25, 2008 at 8:44

community wiki
taleinat

---

1

You can use the django content framework

I made an example of how to implement you models here ->

https://github.com/jmg/django_content_types_example/blob/master/generic_models/models.py

And here you can see how to use the orm ->

https://github.com/jmg/django_content_types_example/blob/master/generic_models/tests.py

Share
Improve this answer
Follow

answered Nov 14, 2013 at 20:41

community wiki
user848192