django-generic-m2m Documentation

Release 0.2.1

charles leifer

Jun 13, 2023

Contents

relate anything to anything. behind the scenes the app uses a table containing two generic foreign keys.

CHAPTER 1

why would I use this?

the purpose of this project is to allow you to create database-level relationships between various objects using a consistent api.

Contents:

1.1 Getting started

1.1.1 Installation

You can pip install django-generic-m2m:

```
pip install django-generic-m2m
```

Alternatively, you can use the version hosted on GitHub, which may contain new or undocumented features:

```
git clone git://github.com/coleifer/django-generic-m2m.git
cd django-generic-m2m
python setup.py install
```

1.1.2 Adding to your Django Project

After installing, adding genericm2m to your projects is a snap. First, add it to your projects' *INSTALLED_APPS* and run *django-admin.py syncdb*:

```
# settings.py
INSTALLED_APPS = [
    ...
    'genericm2m'
]
```

1.1.3 Up and running stupid fast

You need to add a `RelatedObjectsDescriptor` to any model you intend to relate objects from. For example, a news site may want to relate its news stories to various other models:

```
from django.db import models

from genericm2m.models import RelatedObjectsDescriptor


class Story(models.Model):
    # ... story fields ...

    related = RelatedObjectsDescriptor()

    # rest of model definition follows
```

Now you can relate your stories to other objects:

```
>>> story.related.connect(some_city) # create a relationship between story and some_
↪city
>>> story.related.connect(some_public_figure) # ... between story and some_public_
↪figure
```

These relationships can be queried:

```
>>> story.related.all() # find out what "story" has been related to
[<RelatedObject: story related to some_city ("")>,
 <RelatedObject: story related to some_public_figure ("")>]
```

And you can use a custom method on the `QuerySet` to get at those related objects using an optimized query:

```
>>> story.related.all().generic_objects() # traverse the GFK to get the actual objects
[<City: some_city>, <Person: some_public_figure>]
```

Monkeypatching

If the model definition isn't accessible, whether because it is in a 3rd party app or because it is in a contrib app, you can monkeypatch:

```
from django.contrib.auth.models import User

from genericm2m.utils import monkey_patch

monkey_patch(User, 'related')
```

Now you can create relationships from `User` objects:

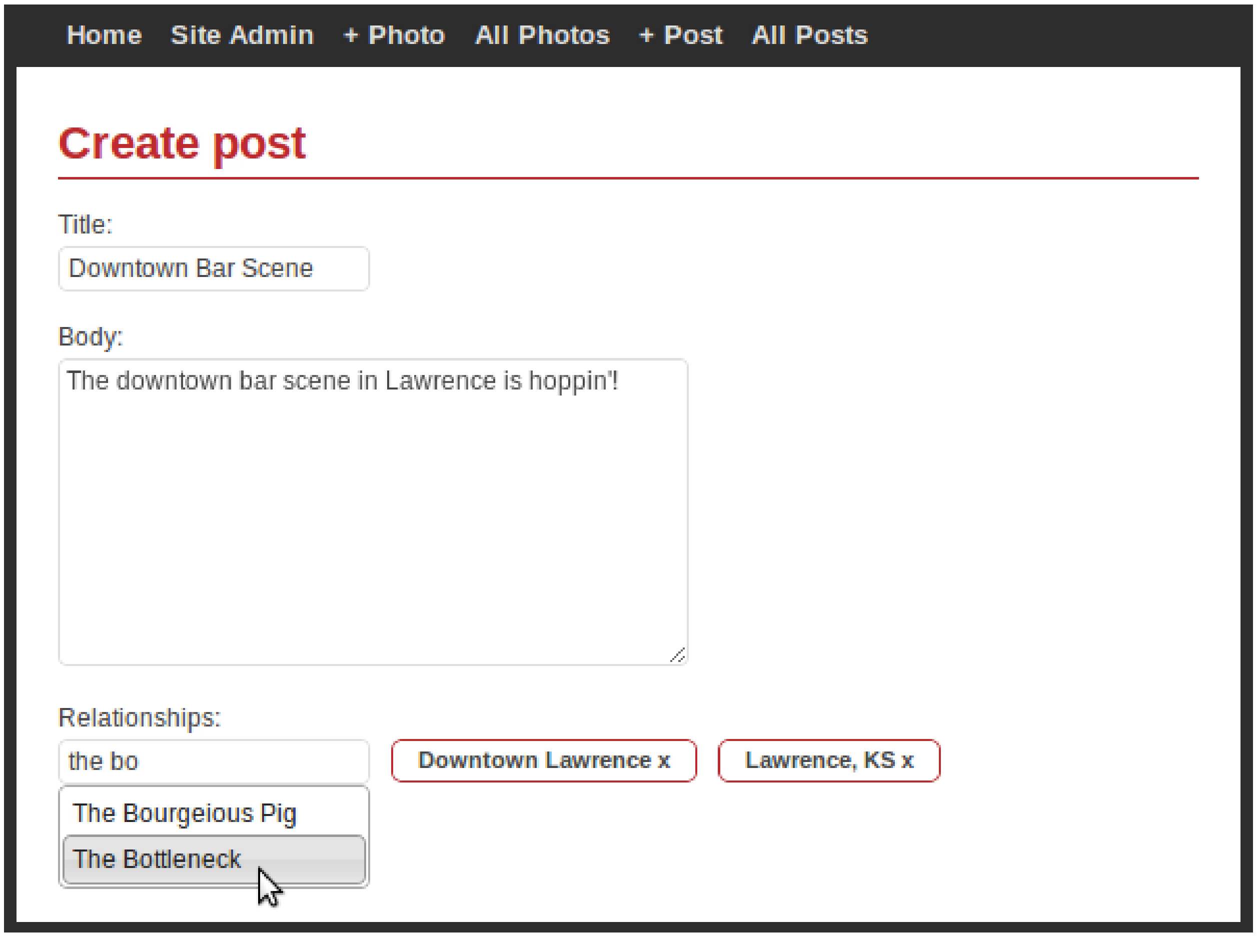
```
>>> some_guy = User.objects.get(username='some_guy') # get a user object
>>> pizza = Food.objects.get(name='pizza') # get a food object
>>> some_guy.related.connect(pizza) # connect the user to the food
```

1.2 Example App

The example app demonstrates how you can use django-generic-m2m to create “tags” between different types of models.

It uses several apps from django basic apps to provide some various content models. Then it uses django-completion to allow users to “autocomplete” various objects in the database, making it easy for users to tag one piece of content with other content from the database.

Below is a screen-shot of a user creating a new blog post. The “relationships” text input does autocompletion making it easy to add “tags” to various models. When the form is submitted, those “tags” become stored using the generic-m2m API.



Downtown Bar Scene

The downtown bar scene in Lawrence is hoppin'!

the bo

1.2.1 How to run the example app

The example app is bundled with django-generic-m2m, but running it requires several external dependencies. For this reason, I’d recommend running it in its own dedicated virtualenv:

```
virtualenv --no-site-packages genericm2m-example
cd genericm2m-example
source bin/activate
```

Now install the latest version of django-generic-m2m from github:

```
pip install -e git+git://github.com/coleifer/django-generic-m2m.git#egg=genericm2m
```

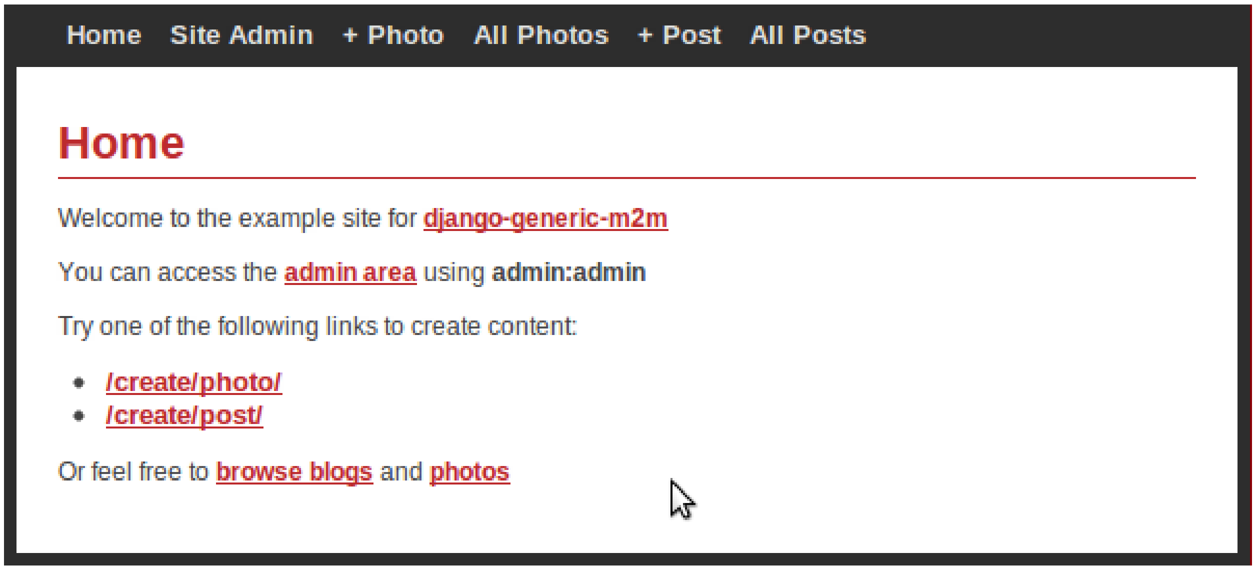
You should see a few lines of text followed by "Successfully installed genericm2m". Now you'll need to install the example app dependencies:

```
pip install -r src/genericm2m/example/requirements.txt
```

This will install the 1.3.X branch of django, django-basic-apps, and django-completion. Once these are installed you are ready to run the example:

```
cd src/genericm2m/example
./manage.py runserver
```

Now navigate to http://127.0.0.1:8000/ in your browser and you will see the example app's homepage:



Home

Welcome to the example site for django-generic-m2m

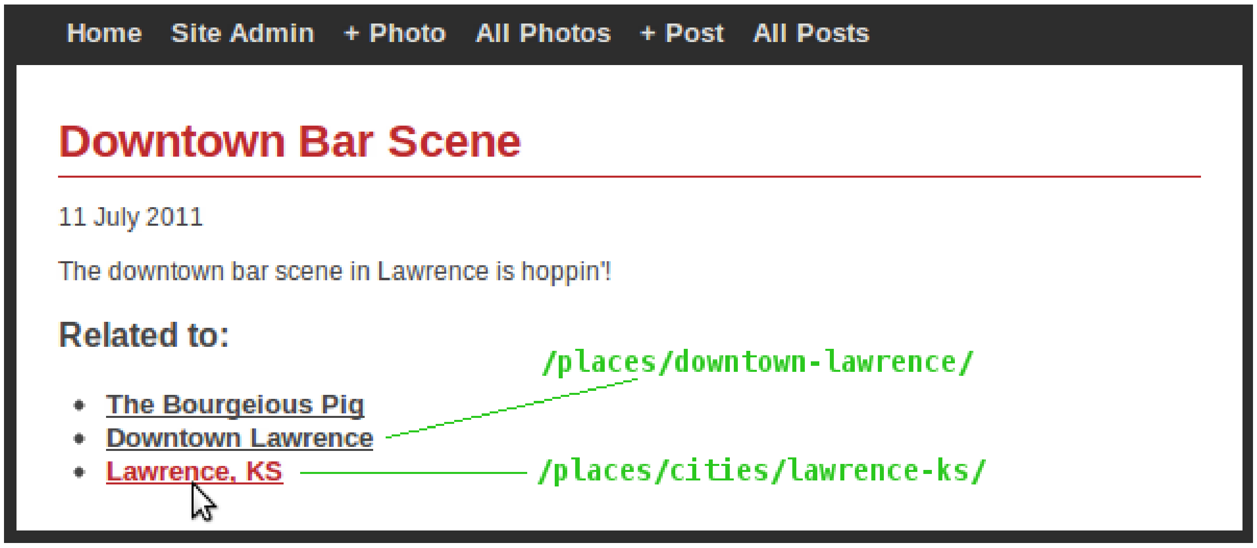
You can access the admin area using admin:admin

Try one of the following links to create content:

- /create/photo/
- /create/post/

Or feel free to browse blogs and photos

If you want to see examples of "model tagging", browse the photos or blogs. There is a section titled "Related to" with links to whatever the object was tagged with:



Downtown Bar Scene

11 July 2011

The downtown bar scene in Lawrence is hoppin'!

Related to:

- The Bourgeious Pig
- Downtown Lawrence
- Lawrence, KS

I'd encourage you to click around, create a few posts or photos and try tagging them with various models.

1.2.2 What is in the example app?

The example app is centered around a few small pieces:

- custom form classes and views to handle creating the relationships
- javascript that handles autocompletion and storing data in the form
- autocomplete providers that make it possible to do autocompletion on our models
- code in template to show the related objects for a post or photo

We'll tackle this stuff one bit at a time starting with the form classes and views, since thats all normal django stuff we're all probably familiar with.

Forms and views

If you open up example/site_app/forms.py in your favorite editor, you'll see a normal `ModelForm` subclass which has a couple additional fields on it:

```
from django.contrib.contenttypes.models import ContentType


class BaseRelationshipsForm(forms.ModelForm):
    relationships = forms.CharField(required=False)
    hidden_relationships = forms.CharField(required=False, widget=forms.HiddenInput())

    def clean_hidden_relationships(self):
        hidden = self.cleaned_data.get('hidden_relationships') or ''

        cts_and_ids = [ct_id for ct_id in hidden.split(',') if ct_id.strip()]
        objects = []

        for ct_id in cts_and_ids:
            content_type_id, object_id = ct_id.split(':')

            ctype = ContentType.objects.get_for_id(int(content_type_id))
            obj = ctype.model_class()._default_manager.get(pk=object_id)

            objects.append(obj)

        return objects
```

This subclass will be used to implement a `generic-m2m`-aware `ModelForm` for blog posts and photos. As you can see from the clean_hidden_relationships method, all we're doing is deserializing a comma-separated list of content-type/primary-key pairs and returning a list of actual objects.

Here's what the code for the `Photo` form class looks like... Nothing too weird – it uses a mixin to auto-generate the slug upon save, but other than that pretty plain-jane:

```
class PhotoForm(BaseRelationshipsForm, SlugifyMixin):
    class Meta:
        model = Photo
        fields = ('title', 'photo',)
```

These forms are used by two views which handle displaying a template and, if everything looks good, creating a new object. The interesting part is right after the initial model save where the newly-created objects gets connected to whatever objects it was tagged with:

```
def generic_completion_view(request, form_class, template):
    form = form_class(request.POST or None, request.FILES or None)

    if request.method == 'POST' and form.is_valid():
        # save the new object instance
        new_obj = form.save()

        # grab the related objects from the form and add them
        # to the new post instance
        for obj in form.cleaned_data['hidden_relationships']:
            new_obj.related.connect(obj)

        return redirect(new_obj)

    return render_to_response(template, {'form': form},
        context_instance=RequestContext(request))

def create_photo(request):
    return generic_completion_view(request, PhotoForm, 'media/create_photo.html')
```

Some JavaScript

On the client-side, we need to do three things:

1. fetch data from our autocomplete view when the user types into the relationships input
2. **upon selecting an item, update a hidden field so the form on the server-side can figure** out what objects we're talking about
3. provide a mechanism for removing previously selected objects

These tasks are accomplished by using jQuery UI's autocomplete widget. The trick I used is cribbed from django-basic-apps, wherein the id of the object selected is stored in the hash of the link to "remove" that object from the list selected items. So you end up with a hidden input full of any number of identifiers, and links with a generic listener that removes the id in question from the hidden input.

Autocomplete providers

django-completion (shameless plug) is an attempt at simplifying the process of providing autocompletion for a set of models. I used it to enable autocompletion on a handful of models from django-basic-apps. The process should look familiar if you've created custom `ModelAdmin` classes before. Here's a representative example:

```
from completion import site, DjangoModelProvider

from basic.blog.models import Post
# ... other imports ...


class PostProvider(DjangoModelProvider):
    def get_title(self, obj):
        return obj.title

    def get_pub_date(self, obj):
        return obj.publish

    def get_data(self, obj):
```

(continues on next page)

(continued from previous page)

```
        return {
            'title': obj.title,
            'url': obj.get_absolute_url(),
        }

# ... other providers ...

site.register(Post, PostProvider)
```

Signal handlers ensure that the autocomplete data is kept fresh whenever a model instance is saved or deleted.

Template code

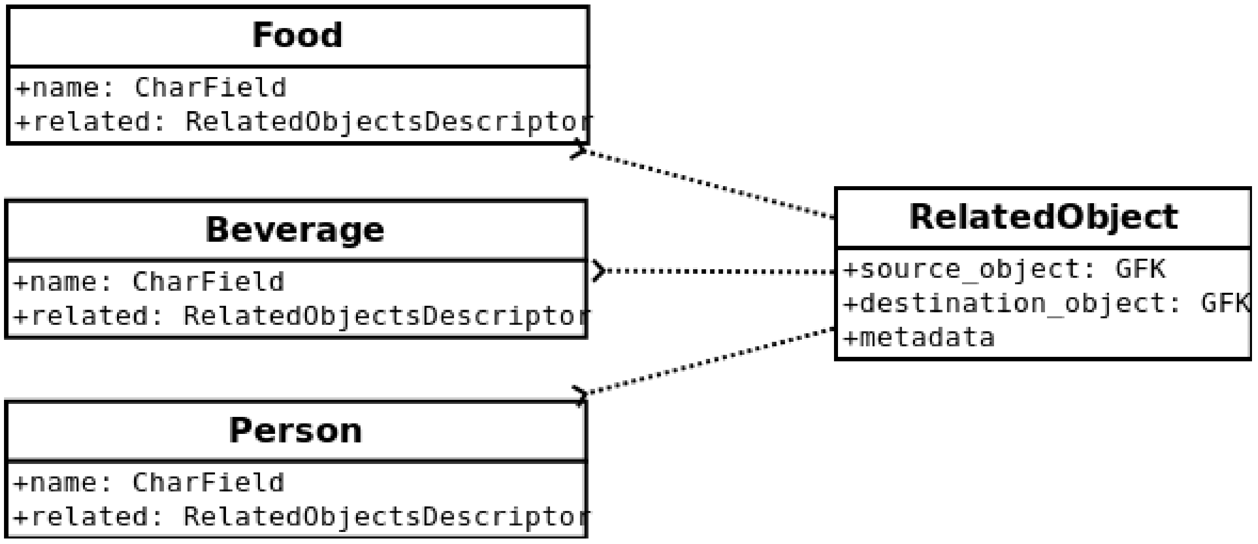
If you look in the template code, all we do is loop over the relationships of the object. The template uses an optimized lookup to traverse the GFK relationships by calling `generic_objects()`. This returns the actual objects that the blog post is connected to.

```
<h3>Related to:</h3>
<ul>
  {% for obj in object.related.all.generic_objects %}
    <li><a href="{{ obj.get_absolute_url }}">{{ obj }}</a></li>
  {% empty %}
    <li>Nothing here</li>
  {% endfor %}
</ul>
```

And that about wraps it up!

1.3 Overview

What its all about is connecting models together and, if you want, creating some metadata about the meaning of that relationship (i.e. a tag).



To this end, django-generic-m2m does three things to make this behavior easier:

1. wraps up all querying and connecting logic in a single attribute that acts on both model instances and the model class
2. allows any model to be used as the intermediary "through" model
3. provides an optimized lookup when GenericForeignKeys are used

1.3.1 Adding to a model

Before you start creating relationships, you'll need to add a RelatedObjectsDescriptor to any model you plan on relating to other models.

Here's a quick example:

```
from django.db import models

from genericm2m.models import RelatedObjectsDescriptor


class Food(models.Model):
    name = models.CharField(max_length=255)

    related = RelatedObjectsDescriptor()

    def __unicode__(self):
        return self.name


class Beverage(models.Model):
    name = models.CharField(max_length=255)

    related = RelatedObjectsDescriptor()

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

If you'd like to add relationships to a model that you don't control (for example the User model from django.contrib.auth), you can use the monkey_patch utility:

```
from django.contrib.auth.models import User

from genericm2m.utils import monkey_patch

monkey_patch(User, name='related')
```

1.3.2 What is this "related" attribute?

The "related" attribute from the previous examples is the way the generic many-to-many is exposed for each model. Behind-the-scenes it is using genericm2m.models.BaseGFKRelatedObject, which looks like this:

```
class BaseGFKRelatedObject(models.Model):
    """
    A generic many-to-many implementation where diverse objects are related
    across a single model to other diverse objects -> using a dual GFK
    """
    # SOURCE OBJECT:
```

(continues on next page)

(continued from previous page)

```
    parent_type = models.ForeignKey(ContentType, related_name="child_%(class)s")
    parent_id = models.IntegerField(db_index=True)
    parent = GenericForeignKey(ct_field="parent_type", fk_field="parent_id")

    # ACTUAL RELATED OBJECT:
    object_type = models.ForeignKey(ContentType, related_name="related_%(class)s")
    object_id = models.IntegerField(db_index=True)
    object = GenericForeignKey(ct_field="object_type", fk_field="object_id")

    class Meta:
        abstract = True
```

There's not really too much that should be weird about this model. It contains two `GenericForeignKeys`, one to represent the "from" object, the source of the connection, and another to represent to "to" object (what "from" is being connected with).

Because "abstract" models cannot store actual objects, the project comes with a default implementation which has two additional fields, `alias` and `creation_date`:

```
class RelatedObject(BaseGFKRelatedObject):
    """
    A subclass of BaseGFKRelatedObject which adds two fields used for tracking
    some metadata about the relationship, an alias and the date the relationship
    was created
    """
    alias = models.CharField(max_length=255, blank=True)
    creation_date = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ('-creation_date',)

    def __unicode__(self):
        return '%s related to %s ("%s")' % (self.parent, self.object, self.alias)
```

1.3.3 Creating and querying relationships

A custom model manager is exposed on each model via the `RelatedObjectsDescriptor`. The API for creating and querying relationships is exposed via this descriptor.

Here is a sample interactive terminal session:

```
>>> # create a handful of objects to use in our demo
>>> pizza = Food.objects.create(name='pizza')
>>> cereal = Food.objects.create(name='cereal')
>>> beer = Beverage.objects.create(name='beer')
>>> soda = Beverage.objects.create(name='soda')
>>> milk = Beverage.objects.create(name='milk')
>>> healthy_eater = User.objects.create_user('healthy_eater', 'healthy@health.com',
↪'secret')
>>> chocula = User.objects.create_user('chocula', 'chocula@postcereal.com', 'garlic')
```

Now that we have some Food, Beverage and User objects, create some connections between them:

```
>>> rel_obj = pizza.related.connect(beer, alias='Beer and pizza are good')
>>> type(rel_obj) # what did we just create?
<class 'genericm2m.models.RelatedObject'>
```

The object that represents the connection is an instance of whatever is passed to the `RelatedObjectDescriptor` when it is added to a model, but the default is `genericm2m.models.RelatedObject`. Here are the interesting properties of the new related object:

```
>>> rel_obj.parent
<Food: pizza>
>>> rel_obj.object
<Beverage: beer>
>>> rel_obj.alias
'Beer and pizza are good'
```

These relationships can be queried:

```
>>> pizza.related.all() # find all objects that pizza has been related to
[<RelatedObject: pizza related to beer ("Beer and pizza are good")>]
```

When the *RelatedObject* is a GFK, as is the case here, the `RelatedObjectsDescriptor` will return a special `QuerySet` class that provides an optimized lookup of any GFK-ed objects:

```
>>> type(pizza.related.all())
<class 'genericm2m.models.GFKOptimizedQuerySet'>
>>> pizza.related.all().generic_objects() # traverse the GFK relationships
[<Beverage: beer>]
```

If the object on the back-side of the relationship also has a `RelatedObjectsDescriptor` with the same intermediary model, reverse lookups are possible:

```
>>> beer.related.related_to() # query the back-side of the relationship
[<RelatedObject: pizza related to beer ("Beer and pizza are good")>]
```

Create some more connections - any combination of models can be used. Below I'm connectiong a Food (cereal) to both Beverage objects (milk) and User objects (Chocula):

```
>>> cereal.related.connect(milk) # connecting to a beverage
<RelatedObject: cereal related to milk ("")>
>>> cereal.related.connect(chocula) # connecting to a user
<RelatedObject: cereal related to chocula ("")>

>>> cereal.related.all() # show what cereal is related to
[<RelatedObject: cereal related to chocula ("")>,
 <RelatedObject: cereal related to milk ("")>]

>>> chocula.related.all() # relationships are ONE WAY
[]
>>> chocula.related.related_to() # querying the backside shows what has been␣
↪connected to chocula
[<RelatedObject: cereal related to chocula ("")>]
```

Also worth noting is that the `RelatedObjectsDescriptor` works on both the instance-level and the class-level, so if we wanted to see all objects related to foods:

```
>>> Food.related.all() # anything that has been related to a food
[<RelatedObject: cereal related to chocula ("")>,
 <RelatedObject: cereal related to milk ("")>,
 <RelatedObject: pizza related to beer ("Beer and pizza are good")>]
```

1.3.4 Using a custom "through" model

It's possible to use a custom "through" model in place of the default `RelatedObject`. If you know you're only going to be using a couple models, this can be a handy way to save queries. Looking at the tests, here's another silly example where we have a `RelatedBeverage` model that our Food model will use:

```
class RelatedBeverage(models.Model):
    food = models.ForeignKey('Food')
    beverage = models.ForeignKey('Beverage')

    class Meta:
        ordering = ('-id',)

class Food(models.Model):
    # ... same as above except for this new attribute:
    related_beverages = RelatedObjectsDescriptor(RelatedBeverage, 'food', 'beverage')
```

The "related_beverages" attribute is an instance of `RelatedObjectsDescriptor`, but it is instantiated with a couple of arguments:

- `RelatedBeverage`: the model to be used to hold the "connections"
- 'food': the field name on the above model which maps to the "from" object
- 'beverage': the field name which maps to the "to" object

Continuing the shell session from above with the same models, foods can be connected to beverages using the new "related_beverages" attribute:

```
>>> pizza.related_beverages.connect(soda)
<RelatedBeverage: RelatedBeverage object>
```

Querying provides the same interface, but since the "to" object is a direct `ForeignKey` to Beverage, a normal django `QuerySet` is used:

```
>>> pizza.related_beverages.all()
[<RelatedBeverage: RelatedBeverage object>]
>>> type(pizza.related_beverages.all())
<class 'django.db.models.query.QuerySet'>
```

A `TypeError` will be raised if you try to connect an invalid object, such as a Person to the "related_beverages":

```
>>> pizza.related_beverages.connect(mario)
*** TypeError: Unable to query ...
```

And lastly, just like before, its possible to query on the class to get all the `RelatedBeverage` objects for our foods:

```
>>> Food.related_beverages.all()
[<RelatedBeverage: RelatedBeverage object>]
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search