How to Add a REST API to an Existing Django Project

This is more an overview than a tutorial. In a tutorial, you'd generally start small and build up the project. This HOWTO is more of a “reading” where we read an existing text to see how it all fits together.

Assume the app you wrote is called “items”, and it's a photogallery tutorial. This is based on the photogallery from a Forcier, Bissex, and Chun book. You should already have done the regular Django REST Framework tutorial, and altered your project settings.py to bring in the REST framework and can alter urls.py to include our urls.py. Then you do this:

```
./manage.py startapp api
cd api
touch serializers.py urls.py
```

At this point, the directory structure should look like this, with the two projects side-by-side:
./mysite *(the root of the project)*
./mysite/items *(the app for which we're adding the API)*
./mysite/api

You start off by making urls for all the models; odds are, you will pare this down.

This example has named urls that differ from the default names. ***It is easier if you use the default names*** – they are here because I needed to figure out which named urls get used in the serializers. The default names would be: `item, item-detail, photo, photo-detail.` You would still need to specify them in urls.py, but not in the serializers.

urls.py:

```
from django.conf.urls import url, include, patterns
import views

apiurls = [
    url(r'^items/$',
        views.ItemListAPIView.as_view(),
        name='items'
    ),
    url(r'^items/(?P<pk>[0-9]+)/$',
        views.ItemAPIView.as_view(),
        name='item-detail'
    ),
    url(r'^photos/$',
        views.PhotoListAPIView.as_view(),
        name='photos'
    ),
    url(r'^photos/(?P<pk>[0-9]+)/$',
        views.PhotoAPIView.as_view(),
        name='photo-detail'
    ),
]
urlpatterns = [
    url(r'^$', views.api_root, name='api-root'),
```

```
    url(r'^', include(apiurls)),
    url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework'))
]
```

Note that this varies from the DRF tutorial in that I suffix all the view class names with APIView. I did this to be explicit, because there could be similar views in the “items” app that have identical names. It's not strictly necessary.

You can then make **api/views.py**:

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.decorators import api_view
from rest_framework.reverse import reverse
from rest_framework import generics
from items.models import Item, Photo
import api.serializers as serializers

class ItemListAPIView(generics.ListCreateAPIView):
    queryset = Item.objects.all()
    serializer_class = serializers.ItemListSerializer

class ItemAPIView(generics.RetrieveUpdateDestroyAPIView):
    queryset = Item.objects.all()
    serializer_class = serializers.ItemSerializer

class PhotoListAPIView(generics.ListCreateAPIView):
    queryset = Photo.objects.all()
    serializer_class = serializers.PhotoListSerializer

class PhotoAPIView(generics.RetrieveUpdateDestroyAPIView):
    queryset = Photo.objects.all()
    serializer_class = serializers.PhotoSerializer

@api_view(('GET',))
def api_root(request, format=None):
    return Response({
        'items': reverse('items', request=request, format=format),
        'photos': reverse('photos', request=request, format=format),
    })
```

This is pretty straightforward. Each view is a dump of the models, and the models are imported from items.models, not defined within the API.

Next up is the **serializer.py** file, which contains a relation field. I'll explain it afterward.

```
from django.forms import widgets
from rest_framework import serializers
from items.models import Item, Photo

class PhotoListSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Photo
        view_name='photo-detail'

class PhotoSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
```

```
        model = Photo
        view_name='photo-detail'

class ItemListSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Item
        view_name = 'item-detail'

class ItemSerializer(serializers.HyperlinkedModelSerializer):
    #photo_set = serializers.HyperlinkedRelatedField(
    #        many=True, read_only=True, view_name='photo-detail')
    photo_set = PhotoListSerializer(many=True)
    class Meta:
        model = Item
        view_name='item-detail'
```

Before I get into the relation, note that view_name is defined on all the serializers. They don't need to be there, because they happen to match the default view names of “photo-detail” and “item-detail”. The other thing to note is that the view names are all -detail, because they're based on the model's name.

We don't use the “photos” and “items” views for the lists. This was confusing to me, at first. These names don't say “this is my name”; they mean “the view name for the model I'm serializing is this.” So, even if we're getting a list of Items or Photos, we're going to create links to the objects, not links to lists of objects.

The separate serializers for the lists and details seem redundant, because, here, they are, but when it's time to improve the API, it'll be nice to have separate lists and details. (For example, in the lists, you only want summary information, like thumbnails.)

The ItemSerializer has a related field. In this sample, it's coded up as a PhotoListSerializer(), which causes the photo's detail to be included in the list. The commented-out code shows a different way to define the related filed, which shows only the photo's url.

Note that the field name is the default Django name for the queryset manager that Django creates for relations. There's a field in Photo called “item”, which is a ForeignKey, and that causes “photo_set” to be created in Item objects. “photo_set” is a Manager that can get records related to the Item.

Here's the code for **items/models.py**, edited to show just the fields, and altered from the tutorial a little:

```
from django.db import models
from django.core.urlresolvers import reverse

# Create your models here.

class Item(models.Model):
    name = models.CharField(max_length=250)
    description = models.TextField()

class Photo(models.Model):
    item = models.ForeignKey(Item)
    title = models.CharField(max_length=100)
```

```
    image = models.ImageField(upload_to='photos')
    caption = models.CharField(max_length=250, blank=True)
```

The ForeignKey relation is boldfaced. This field causes “photo_set” to be created on every Item instance.

In this situation, I used the default name so that I didn't have to set the related_name for the field. I figured that there might be situations where the app code can't be changed, so I have to go with the defaults.

So, there you go, a quick API add-on. It took me a few hours to figure this out and write it, but now it should only take you a few minutes to get up and running. The real work comes in refining the API so it's useful.

Comments to johnk@riceball.com