Tastypie Documentation

Release 0.8.3-beta

Daniel Lindsley, Cody Soyland & Matt Croydon

August 03, 2013

CONTENTS

Tastypie is an webservice API framework for Django. It provides a convenient, yet powerful and highly customizable, abstraction for creating REST-style interfaces.

CHAPTER

ONE

GETTING STARTED WITH TASTYPIE

Tastypie is a reusable app (that is, it relies only on it's own code and focuses on providing just a REST-style API) and is suitable for providing an API to any application without having to modify the sources of that app.

Not everyone's needs are the same, so Tastypie goes out of it's way to provide plenty of hooks for overridding or extending how it works.

Note: If you hit a stumbling block, you can join #tastypie on irc.freenode.net to get help.

This tutorial assumes that you have a basic understand of Django as well as how proper REST-style APIs ought to work. We will only explain the portions of the code that are Tastypie-specific in any kind of depth.

For example purposes, we'll be adding an API to a simple blog application. Here is `myapp/models.py`:

```
import datetime
from django.contrib.auth.models import User
from django.db import models
from django.template.defaultfilters import slugify


class Entry(models.Model):
    user = models.ForeignKey(User)
    pub_date = models.DateTimeField(default=datetime.datetime.now)
    title = models.CharField(max_length=200)
    slug = models.SlugField
    body = models.TextField()

    def __unicode__(self):
        return self.title

    def save(self, *args, **kwargs):
        # For automatic slug generation.
        if not self.slug:
            self.slug = slugify(self.title)[:50]

        return super(Entry, self).save(*args, **kwargs)
```

With that, we'll move on to installing and configuring Tastypie.

1.1 Installation

Installing Tastypie is as simple as checking out the source and adding it to your project or `PYTHONPATH`.

1. Download the dependencies:
   - Python 2.4+
   - Django 1.0+ (tested on Django 1.1+)
   - `mimeparse` (http://code.google.com/p/mimeparse/)
   - `dateutil` (http://labix.org/python-dateutil)
   - **OPTIONAL** - `lxml` (http://codespeak.net/lxml/) if using the XML serializer
   - **OPTIONAL** - `pyyaml` (http://pyyaml.org/) if using the YAML serializer
   - **OPTIONAL** - `uuid` (present in 2.5+, downloadable from http://pypi.python.org/pypi/uuid/) if using the `ApiKey` authentication
2. Check out tastypie from GitHub.
3. Either symlink the `tastypie` directory into your project or copy the directory in. What ever works best for you.

Note: Once tastypie passes version 1.0, it will become officially available on PyPI. Once that is the case, a `sudo pip install tastypie` or `sudo easy_install tastypie` should be available.

1.2 Configuration

The only mandatory configuration is adding `'tastypie'` to your `INSTALLED_APPS`. This isn't strictly necessary, as Tastypie has only one non-required model, but may ease usage.

You have the option to set up a number of settings (see *Tastypie Settings*) but most have sane defaults and are not required unless you need to tweak their values.

1.3 Creating Resources

REST-style architecture talks about resources, so unsurprisingly integrating with Tastypie involves creating `Resource` classes. For our simple application, we'll create a file for these in `myapp/api.py`, though they can live anywhere in your application:

```
# myapp/api.py
from tastypie.resources import ModelResource
from myapp.models import Entry


class EntryResource(ModelResource):
    class Meta:
        queryset = Entry.objects.all()
        resource_name = 'entry'
```

This class, by virtue of being a `ModelResource` subclass, will introspect all non-relational fields on the `Entry` model and create it's own `ApiFields` that map to those fields, much like the way Django's `ModelForm` class introspects.

Note: The `resource_name` within the `Meta` class is optional. If not provided, it is automatically generated off the classname, removing any instances of `Resource` and lowercasing the string. So `EntryResource` would become just `entry`.

It's included in this example for clarity, especially when looking at the URLs, but you may feel free to omit it if you're comfortable with this behavior.

1.4 Hooking Up The Resource(s)

Now that we have our `EntryResource`, we can hook it up in our URLconf. To do this, we simply instantiate the resource in our URLconf and hook up its `urls`:

```
# urls.py
from django.conf.urls.defaults import *
from myapp.api import EntryResource

entry_resource = EntryResource()

urlpatterns = patterns('',
    # The normal jazz here...
    (r'^blog/', include('myapp.urls')),
    (r'^api/entry/', include(entry_resource.urls)),
)
```

Now it's just a matter of firing up server (`./manage.py runserver`) and going to http://127.0.0.1:8000/api/entry/?format=json. You should get back a list of `Entry`-like objects.

Note: The `?format=json` is an override required to make things look decent in the browser (accept headers vary between browsers). Tastypie properly handles the `Accept` header. So the following will work properly:

```
curl -H "Accept: application/json" http://127.0.0.1:8000/api/entry/
```

But if you're sure you want something else (or want to test in a browser), Tastypie lets you specify `?format=...` when you really want to force a certain type.

At this point, a bunch of other URLs are also available. Try out any/all of the following (assuming you have at least three records in the database):

- http://127.0.0.1:8000/api/entry/?format=json
- http://127.0.0.1:8000/api/entry/1/?format=json
- http://127.0.0.1:8000/api/entry/schema/?format=json
- http://127.0.0.1:8000/api/entry/set/1;3/?format=json

With just seven lines of code, we have a full working REST interface to our `Entry` model. In addition, full GET/POST/PUT/DELETE support is already there, so it's possible to really work with all of the data. Well, *almost*.

You see, you'll note that not quite all of our data is there. Markedly absent is the `user` field, which is a `ForeignKey` to Django's `User` model. Tastypie does **NOT** introspect related data because it has no way to know how you want to represent that data.

And since that relation isn't there, any attempt to POST/PUT new data will fail, because no `user` is present, which is a required field on the model.

This is easy to fix, but we'll need to flesh out out API a little more.

1.5 Creating More Resources

In order to handle our `user` relation, we'll need to create a `UserResource` and tell the `EntryResource` to use it. So we'll modify `myapp/api.py` to match the following code:

```
# myapp/api.py
from django.contrib.auth.models import User
from tastypie import fields
from tastypie.resources import ModelResource
from myapp.models import Entry


class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
        resource_name = 'user'


class EntryResource(ModelResource):
    user = fields.ForeignKey(UserResource, 'user')

    class Meta:
        queryset = Entry.objects.all()
        resource_name = 'entry'
```

We simply created a new `ModelResource` subclass called `UserResource`. Then we added a field to `EntryResource` that specified that the `user` field points to a `UserResource` for that data.

Now we should be able to get all of the fields back in our response. But since we have another full, working resource on our hands, we should hook that up to our API as well. And there's a better way to do it.

1.6 Adding To The Api

Tastypie ships with an `Api` class, which lets you bind multiple `Resources` together to form a coherent API. Adding it to the mix is simple.

We'll go back to our URLconf (`urls.py`) and change it to match the following:

```
# urls.py
from django.conf.urls.defaults import *
from tastypie.api import Api
from myapp.api import EntryResource, UserResource

v1_api = Api(name='v1')
v1_api.register(UserResource())
v1_api.register(EntryResource())

urlpatterns = patterns('',
    # The normal jazz here...
    (r'^blog/', include('myapp.urls')),
    (r'^api/', include(v1_api.urls)),
)
```

Note that we're now creating an `Api` instance, registering our `EntryResource` and `UserResource` instances with it and that we've modified the urls to now point to `v1_api.urls`.

This makes even more data accessible, so if we start up the `runserver` again, the following URLs should work:

- http://127.0.0.1:8000/api/v1/?format=json
- http://127.0.0.1:8000/api/v1/user/?format=json
- http://127.0.0.1:8000/api/v1/user/1/?format=json
- http://127.0.0.1:8000/api/v1/user/schema/?format=json
- http://127.0.0.1:8000/api/v1/user/set/1;3/?format=json
- http://127.0.0.1:8000/api/v1/entry/?format=json
- http://127.0.0.1:8000/api/v1/entry/1/?format=json
- http://127.0.0.1:8000/api/v1/entry/schema/?format=json
- http://127.0.0.1:8000/api/v1/entry/set/1;3/?format=json

Additionally, the representations out of `EntryResource` will now include the `user` field and point to an endpoint like `/api/v1/users/1/` to access that user's data. And full POST/PUT delete support should now work.

But there's several new problems. One is that our new `UserResource` leaks too much data, including fields like `email`, `password`, `is_active` and `is_staff`. Another is that we may not want to allow end users to alter `User` data. Both of these problems are easily fixed as well.

1.7 Limiting Data And Access

Cutting out the `email`, `password`, `is_active` and `is_staff` fields is easy to do. We simply modify our `UserResource` code to match the following:

```
class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
        resource_name = 'user'
        excludes = ['email', 'password', 'is_active', 'is_staff', 'is_superuser']
```

The `excludes` directive tells `UserResource` which fields not to include in the output. If you'd rather whitelist fields, you could do:

```
class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
        resource_name = 'user'
        fields = ['username', 'first_name', 'last_name', 'last_login']
```

Now that the undesirable fields are no longer included, we can look at limiting access. This is also easy and involves making our `UserResource` look like:

```
class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
        resource_name = 'user'
        excludes = ['email', 'password', 'is_active', 'is_staff', 'is_superuser']
        allowed_methods = ['get']
```

Now only HTTP GET requests will be allowed on `/api/v1/user/` endpoints. If you require more granular control, both `list_allowed_methods` and `detail_allowed_methods` options are supported.

1.8 Beyond The Basics

We now have a full working API for our application. But Tastypie supports many more features, like:

- `authentication`
- `caching`
- `throttling`
- `filtering_sorting`
- `serialization`

Tastypie is also very easy to override and extend. For some common patterns and approaches, you should refer to the `cookbook` documentation.

CHAPTER

TWO

TASTYPIE SETTINGS

This is a comprehensive list of the settings Tastypie recognizes.

2.1 API_LIMIT_PER_PAGE

Optional

This setting controls what the default number of records Tastypie will show in a list view is.

This is only used when a user does not specify a `limit` GET parameter and the `Resource` subclass has not overridden the number to be shown.

An example:

```
API_LIMIT_PER_PAGE = 50
```

Defaults to 20.