Tastypie Documentation

Release 0.14.0

Daniel Lindsley & the Tastypie core team

Jul 03, 2017

Contents

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

CHAPTER 1

Getting Started with Tastypie

Tastypie is a reusable app (that is, it relies only on its 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 its way to provide plenty of hooks for overriding 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 understanding 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`:

```
from tastypie.utils.timezone import now
from django.contrib.auth.models import User
from django.db import models
from django.utils.text import slugify


class Entry(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    pub_date = models.DateTimeField(default=now)
    title = models.CharField(max_length=200)
    slug = models.SlugField(null=True, blank=True)
    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.

Installation

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

1. Download the dependencies:

- Python 2.7+ or Python 3.4+
- Django 1.8+
- `python-mimeparse` 0.1.4+ (http://pypi.python.org/pypi/python-mimeparse)
- `dateutil` (http://labix.org/python-dateutil)
- **OPTIONAL** - `lxml` (http://lxml.de/) and `defusedxml` (https://pypi.python.org/pypi/defusedxml) if using the XML serializer
- **OPTIONAL** - `pyyaml` (http://pyyaml.org/) if using the YAML serializer

2. Either check out tastypie from GitHub or to pull a release off PyPI. Doing `sudo pip install django-tastypie` or `sudo easy_install django-tastypie` is all that should be required.
3. Either symlink the `tastypie` directory into your project or copy the directory in. What ever works best for you.

Configuration

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

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

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 its 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`.

We've included the `resource_name` attribute in this example for clarity, especially when looking at the URLs, but you should feel free to omit it if you're comfortable with the automatic behavior.

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 import url, include
from myapp.api import EntryResource

entry_resource = EntryResource()

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

However, if you try sending a POST/PUT/DELETE to the resource, you find yourself getting "401 Unauthorized" errors. For safety, Tastypie ships with the `authorization` class ("what are you allowed to do") set to `ReadOnlyAuthorization`. This makes it safe to expose on the web, but prevents us from doing POST/PUT/DELETE. Let's enable those:

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


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

Warning: This is now great for testing in development but **VERY INSECURE**. You should never put a `Resource` like this out on the internet. Please spend some time looking at the authentication/authorization classes available in Tastypie.

With just nine 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 our API a little more.

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):
    # Maps `Entry.user` to a Tastypie `ForeignKey` field named `user`,
    # which gets serialized using `UserResource`. The first appearance of
    # 'user' on the next line of code is the Tastypie field name, the 2nd
    # appearance tells the `ForeignKey` it maps to the `user` attribute of
    # `Entry`. Field names and model attributes don't have to be the same.
    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.

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 import url, include
from tastypie.api import Api
from myapp.api import EntryResource, UserResource

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

urlpatterns = [
    # The normal jazz here...
    url(r'^blog/', include('myapp.urls')),
    url(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.

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.

Beyond The Basics

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

- *Authentication*
- *Authorization*
- *Caching*
- *Throttling*
- *Resources* (filtering & sorting)
- *Serialization*

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

CHAPTER 2

Interacting With The API

Now that you've got a shiny new REST-style API in place, let's demonstrate how to interact with it. We'll assume that you have cURL installed on your system (generally available on most modern Mac & Linux machines), but any tool that allows you to control headers & bodies on requests will do.

We'll assume that we're interacting with the following Tastypie code:

```
# myapp/api/resources.py
from django.contrib.auth.models import User
from tastypie.authorization import Authorization
from tastypie import fields
from tastypie.resources import ModelResource, ALL, ALL_WITH_RELATIONS
from myapp.models import Entry


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


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

    class Meta:
        queryset = Entry.objects.all()
        resource_name = 'entry'
        authorization = Authorization()
        filtering = {
            'user': ALL_WITH_RELATIONS,
            'pub_date': ['exact', 'lt', 'lte', 'gte', 'gt'],
        }
```

```
# urls.py
from django.conf.urls import url, include
from tastypie.api import Api
from myapp.api.resources import EntryResource, UserResource

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

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

Let's fire up a shell & start exploring the API!

Front Matter

Tastypie tries to treat all clients & all serialization types as equally as possible. It also tries to be a good 'Net citizen & respects the HTTP method used as well as the `Accepts` headers sent. Between these two, you control all interactions with Tastypie through relatively few endpoints.

> **Warning:** Should you try these URLs in your browser, be warned you **WILL** need to append `?format=json` (or `xml` or `yaml`) to the URL. Your browser requests `application/xml` before `application/json`, so you'll always get back XML if you don't specify it.
>
> That's also why it's recommended that you explore via curl, because you avoid your browser's opinionated requests & get something closer to what any programmatic clients will get.

Fetching Data

Since reading data out of an API is a very common activity (and the easiest type of request to make), we'll start there. Tastypie tries to expose various parts of the API & interlink things within the API (HATEOAS).

Api-Wide

We'll start at the highest level:

```
curl http://localhost:8000/api/v1/
```

You'll get back something like:

```
{
    "entry": {
        "list_endpoint": "/api/v1/entry/",
        "schema": "/api/v1/entry/schema/"
    },
```

```
    "user": {
        "list_endpoint": "/api/v1/user/",
        "schema": "/api/v1/user/schema/"
    }
}
```

This lists out all the different `Resource` classes you registered in your URLconf with the API. Each one is listed by the `resource_name` you gave it and provides the `list_endpoint` & the `schema` for the resource.

Note that these links try to direct you to other parts of the API, to make exploration/discovery easier. We'll use these URLs in the next several sections.

To demonstrate another format, you could run the following to get the XML variant of the same information:

```
curl -H "Accept: application/xml" http://localhost:8000/api/v1/
```

To which you'd receive:

```
<?xml version="1.0" encoding="utf-8"?>
<response>
  <entry type="hash">
    <list_endpoint>/api/v1/entry/</list_endpoint>
    <schema>/api/v1/entry/schema/</schema>
  </entry>
  <user type="hash">
    <list_endpoint>/api/v1/user/</list_endpoint>
    <schema>/api/v1/user/schema/</schema>
  </user>
</response>
```

We'll stick to JSON for the rest of this document, but using XML should be OK to do at any time.

It's also possible to get all schemas (*Inspecting The Resource's Schema*) in a single request:

```
curl http://localhost:8000/api/v1/?fullschema=true
```

You'll get back something like:

```
{
    "entry": {
        "list_endpoint": "/api/v1/entry/",
        "schema": {
            "default_format": "application/json",
            "fields": {
                "body": {
                    "help_text": "Unicode string data. Ex: \"Hello World\"",
                    "nullable": false,
                    "readonly": false,
                    "type": "string"
                },
                ...
            },
            "filtering": {
                "pub_date": ["exact", "lt", "lte", "gte", "gt"],
                "user": 2
            }
        }
    },
}
```

Inspecting The Resource's Schema

Since the api-wide view gave us a `schema` URL, let's inspect that next. We'll use the `entry` resource. Again, a simple GET request by curl:

```
curl http://localhost:8000/api/v1/entry/schema/
```

This time, we get back a lot more data:

```
{
    "default_format": "application/json",
    "fields": {
        "body": {
            "help_text": "Unicode string data. Ex: \"Hello World\"",
            "nullable": false,
            "readonly": false,
            "type": "string"
        },
        "id": {
            "help_text": "Unicode string data. Ex: \"Hello World\"",
            "nullable": false,
            "readonly": false,
            "type": "string"
        },
        "pub_date": {
            "help_text": "A date & time as a string. Ex: \"2010-11-10T03:07:43\"",
            "nullable": false,
            "readonly": false,
            "type": "datetime"
        },
        "resource_uri": {
            "help_text": "Unicode string data. Ex: \"Hello World\"",
            "nullable": false,
            "readonly": true,
            "type": "string"
        },
        "slug": {
            "help_text": "Unicode string data. Ex: \"Hello World\"",
            "nullable": false,
            "readonly": false,
            "type": "string"
        },
        "title": {
            "help_text": "Unicode string data. Ex: \"Hello World\"",
            "nullable": false,
            "readonly": false,
            "type": "string"
        },
        "user": {
            "help_text": "A single related resource. Can be either a URI or set of␣
↪nested resource data.",
            "nullable": false,
            "readonly": false,
            "type": "related"
            "related_type": "to_one"
            "related_schema": "/api/v1/user/schema/"
        }
    },
```

```
    "filtering": {
        "pub_date": ["exact", "lt", "lte", "gte", "gt"],
        "user": 2
    }
}
```

This lists out the `default_format` this resource responds with, the `fields` on the resource & the `filtering` options available. This information can be used to prepare the other aspects of the code for the data it can obtain & ways to filter the resources.

Getting A Collection Of Resources

Let's get down to fetching live data. From the api-wide view, we'll hit the `list_endpoint` for `entry`:

```
curl http://localhost:8000/api/v1/entry/
```

We get back data that looks like:

```
{
    "meta": {
        "limit": 20,
        "next": null,
        "offset": 0,
        "previous": null,
        "total_count": 3
    },
    "objects": [{
        "body": "Welcome to my blog!",
        "id": "1",
        "pub_date": "2011-05-20T00:46:38",
        "resource_uri": "/api/v1/entry/1/",
        "slug": "first-post",
        "title": "First Post",
        "user": "/api/v1/user/1/"
    },
    {
        "body": "Well, it's been awhile and I still haven't updated. ",
        "id": "2",
        "pub_date": "2011-05-21T00:46:58",
        "resource_uri": "/api/v1/entry/2/",
        "slug": "second-post",
        "title": "Second Post",
        "user": "/api/v1/user/1/"
    },
    {
        "body": "I'm really excited to get started with this new blog. It's gonna be␣
↪great!",
        "id": "3",
        "pub_date": "2011-05-20T00:47:30",
        "resource_uri": "/api/v1/entry/3/",
        "slug": "my-blog",
        "title": "My Blog",
        "user": "/api/v1/user/2/"
    }]
}
```

Some things to note:

- By default, you get a paginated set of objects (20 per page is the default).
- In the `meta`, you get a `previous` & `next`. If available, these are URIs to the previous & next pages.
- You get a list of resources/objects under the `objects` key.
- Each resources/object has a `resource_uri` field that points to the detail view for that object.
- The foreign key to `User` is represented as a URI by default. If you're looking for the full `UserResource` to be embedded in this view, you'll need to add `full=True` to the `fields.ToOneField`.

If you want to skip paginating, simply run:

```
curl http://localhost:8000/api/v1/entry/?limit=0
```

Be warned this will return all objects, so it may be a CPU/IO-heavy operation on large datasets.

Let's try filtering on the resource. Since we know we can filter on the `user`, we'll fetch all posts by the `daniel` user with:

```
curl http://localhost:8000/api/v1/entry/?user__username=daniel
```

We get back what we asked for:

```
{
    "meta": {
        "limit": 20,
        "next": null,
        "offset": 0,
        "previous": null,
        "total_count": 2
    },
    "objects": [{
        "body": "Welcome to my blog!",
        "id": "1",
        "pub_date": "2011-05-20T00:46:38",
        "resource_uri": "/api/v1/entry/1/",
        "slug": "first-post",
        "title": "First Post",
        "user": "/api/v1/user/1/"
    },
    {
        "body": "Well, it's been awhile and I still haven't updated. ",
        "id": "2",
        "pub_date": "2011-05-21T00:46:58",
        "resource_uri": "/api/v1/entry/2/",
        "slug": "second-post",
        "title": "Second Post",
        "user": "/api/v1/user/1/"
    }]
}
```

Where there were three posts before, now there are only two.

Getting A Detail Resource

Since each resource/object in the list view had a `resource_uri`, let's explore what's there:

```
curl http://localhost:8000/api/v1/entry/1/
```

We get back a similar set of data that we received from the list view:

```
{
    "body": "Welcome to my blog!",
    "id": "1",
    "pub_date": "2011-05-20T00:46:38",
    "resource_uri": "/api/v1/entry/1/",
    "slug": "first-post",
    "title": "First Post",
    "user": "/api/v1/user/1/"
}
```

Where this proves useful (for example) is present in the data we got back. We know the URI of the `User` associated with this blog entry. Let's run:

```
curl http://localhost:8000/api/v1/user/1/
```

Without ever seeing any aspect of the `UserResource` & just following the URI given, we get back:

```
{
    "date_joined": "2011-05-20T00:42:14.990617",
    "first_name": "",
    "id": "1",
    "last_login": "2011-05-20T00:44:57.510066",
    "last_name": "",
    "resource_uri": "/api/v1/user/1/",
    "username": "daniel"
}
```

You can do a similar fetch using the following Javascript/jQuery (though be wary of same-domain policy):

```
$.ajax({
  url: 'http://localhost:8000/api/v1/user/1/',
  type: 'GET',
  accepts: 'application/json',
  dataType: 'json'
})
```

Selecting A Subset Of Resources

Sometimes you may want back more than one record, but not an entire list view nor do you want to do multiple requests. Tastypie includes a "set" view, which lets you cherry-pick the objects you want. For example, if we just want the first & third `Entry` resources, we'd run:

```
curl "http://localhost:8000/api/v1/entry/set/1;3/"
```

Note: Quotes are needed in this case because of the semicolon delimiter between primary keys. Without the quotes, bash tries to split it into two statements. No extraordinary quoting will be necessary in your application (unless your API client is written in bash :D).

And we get back just those two objects:

```
{
    "objects": [{
        "body": "Welcome to my blog!",
        "id": "1",
        "pub_date": "2011-05-20T00:46:38",
        "resource_uri": "/api/v1/entry/1/",
        "slug": "first-post",
        "title": "First Post",
        "user": "/api/v1/user/1/"
    },
    {
        "body": "I'm really excited to get started with this new blog. It's gonna be␣
↪great!",
        "id": "3",
        "pub_date": "2011-05-20T00:47:30",
        "resource_uri": "/api/v1/entry/3/",
        "slug": "my-blog",
        "title": "My Blog",
        "user": "/api/v1/user/2/"
    }]
}
```

Note that, like the list view, you get back a list of `objects`. Unlike the list view, there is **NO** pagination applied to these objects. You asked for them, you're going to get them all.

Sending Data

Tastypie also gives you full write capabilities in the API. Since the `EntryResource` has the no-limits `Authentication` & `Authorization` on it, we can freely write data.

> **Warning:** Note that this is a huge security hole as well. Don't put unauthorized write-enabled resources on the Internet, because someone will trash your data.
>
> This is why `ReadOnlyAuthorization` is the default in Tastypie & why you must override to provide more access.

The good news is that there are no new URLs to learn. The "list" & "detail" URLs we've been using to fetch data *ALSO* support the `POST`/`PUT`/`DELETE` HTTP methods.

Creating A New Resource (POST)

Let's add a new entry. To create new data, we'll switch from `GET` requests to the familiar `POST` request.

Note: Tastypie encourages "round-trippable" data, which means the data you can GET should be able to be POST/PUT'd back to recreate the same object.

If you're ever in question about what you should send, do a GET on another object & see what Tastypie thinks it should look like.

To create new resources/objects, you will `POST` to the list endpoint of a resource. Trying to `POST` to a detail endpoint has a different meaning in the REST mindset (meaning to add a resource as a child of a resource of the same type).

As with all Tastypie requests, the headers we request are important. Since we've been using primarily JSON throughout, let's send a new entry in JSON format:

```
curl --dump-header - -H "Content-Type: application/json" -X POST --data '{"body":
↪"This will prbbly be my lst post.", "pub_date": "2011-05-22T00:46:38", "slug":
↪"another-post", "title": "Another Post", "user": "/api/v1/user/1/"}' http://
↪localhost:8000/api/v1/entry/
```

The `Content-Type` header here informs Tastypie that we're sending it JSON. We send the data as a JSON-serialized body (**NOT** as form-data in the form of URL parameters). What we get back is the following response:

```
HTTP/1.0 201 CREATED
Date: Fri, 20 May 2011 06:48:36 GMT
Server: WSGIServer/0.1 Python/2.7
Content-Type: text/html; charset=utf-8
Location: http://localhost:8000/api/v1/entry/4/
```

You'll also note that we get a correct HTTP status code back (201) & a `Location` header, which gives us the URI to our newly created resource.

Passing `--dump-header -` is important, because it gives you all the headers as well as the status code. When things go wrong, this will be useful information to help with debugging. For instance, if we send a request without a `user`:

```
curl --dump-header - -H "Content-Type: application/json" -X POST --data '{"body":
↪"This will prbbly be my lst post.", "pub_date": "2011-05-22T00:46:38", "slug":
↪"another-post", "title": "Another Post"}' http://localhost:8000/api/v1/entry/
```

We get back:

```
HTTP/1.0 400 BAD REQUEST
Date: Fri, 20 May 2011 06:53:02 GMT
Server: WSGIServer/0.1 Python/2.7
Content-Type: text/html; charset=utf-8

The 'user' field has no data and doesn't allow a default or null value.
```

You can do a similar POST using the following Javascript/jQuery (though be wary of same-domain policy):

```
# This may require the ``json2.js`` library for older browsers.
var data = JSON.stringify({
    "body": "This will prbbly be my lst post.",
    "pub_date": "2011-05-22T00:46:38",
    "slug": "another-post",
    "title": "Another Post"
});

$.ajax({
  url: 'http://localhost:8000/api/v1/entry/',
  type: 'POST',
  contentType: 'application/json',
  data: data,
  dataType: 'json',
  processData: false
})
```

Updating An Existing Resource (PUT)

You might have noticed that we made some typos when we submitted the POST request. We can fix this using a `PUT` request to the detail endpoint (modify this instance of a resource).:

```
curl --dump-header - -H "Content-Type: application/json" -X PUT --data '{"body":
↪"This will probably be my last post.", "pub_date": "2011-05-22T00:46:38", "slug":
↪"another-post", "title": "Another Post", "user": "/api/v1/user/1/"}' http://
↪localhost:8000/api/v1/entry/4/
```

After fixing up the `body`, we get back:

```
HTTP/1.0 204 NO CONTENT
Date: Fri, 20 May 2011 07:13:21 GMT
Server: WSGIServer/0.1 Python/2.7
Content-Length: 0
Content-Type: text/html; charset=utf-8
```

We get a 204 status code, meaning our update was successful. We don't get a `Location` header back because we did the `PUT` on a detail URL, which presumably did not change.

Note: A `PUT` request requires that the entire resource representation be enclosed. Missing fields may cause errors, or be filled in by default values.

Partially Updating An Existing Resource (PATCH)

In some cases, you may not want to send the entire resource when updating. To update just a subset of the fields, we can send a `PATCH` request to the detail endpoint.:

```
curl --dump-header - -H "Content-Type: application/json" -X PATCH --data '{"body":
↪"This actually is my last post."}' http://localhost:8000/api/v1/entry/4/
```

To which we should get back:

```
HTTP/1.0 202 ACCEPTED
Date: Fri, 20 May 2011 07:13:21 GMT
Server: WSGIServer/0.1 Python/2.7
Content-Length: 0
Content-Type: text/html; charset=utf-8
```

Updating A Whole Collection Of Resources (PUT)

You can also, in rare circumstances, update an entire collection of objects. By sending a `PUT` request to the list view of a resource, you can replace the entire collection.

Warning: This deletes all of the objects first, then creates the objects afresh. This is done because determining which objects are the same is actually difficult to get correct in the general case for all people.

Send a request like:

```
curl --dump-header - -H "Content-Type: application/json" -X PUT --data '{"objects": [{
↪"body": "Welcome to my blog!","id": "1","pub_date": "2011-05-20T00:46:38","resource_
↪uri": "/api/v1/entry/1/","slug": "first-post","title": "First Post","user": "/api/
↪v1/user/1/"},{"body": "I'm really excited to get started with this new blog. It's␣
↪gonna be great!","id": "3","pub_date": "2011-05-20T00:47:30","resource_uri": "/api/
↪v1/entry/3/","slug": "my-blog","title": "My Blog","user": "/api/v1/user/2/"}]}'␣
↪http://localhost:8000/api/v1/entry/
```

And you'll get back a response like:

```
HTTP/1.0 204 NO CONTENT
Date: Fri, 20 May 2011 07:13:21 GMT
Server: WSGIServer/0.1 Python/2.7
Content-Length: 0
Content-Type: text/html; charset=utf-8
```

Deleting Data

No CRUD setup would be complete without the ability to delete resources/objects. Deleting also requires significantly less complicated requests than `POST`/`PUT`.

Deleting A Single Resource

We've decided that we don't like the entry we added & edited earlier. Let's delete it (but leave the other objects alone):

```
curl --dump-header - -H "Content-Type: application/json" -X DELETE  http://
↪localhost:8000/api/v1/entry/4/
```

Once again, we get back the "Accepted" response of a 204:

```
HTTP/1.0 204 NO CONTENT
Date: Fri, 20 May 2011 07:28:01 GMT
Server: WSGIServer/0.1 Python/2.7
Content-Length: 0
Content-Type: text/html; charset=utf-8
```

If we request that resource, we get a 404 to show it's no longer there:

```
curl --dump-header - http://localhost:8000/api/v1/entry/4/

HTTP/1.0 404 GONE
Date: Fri, 20 May 2011 07:29:02 GMT
Server: WSGIServer/0.1 Python/2.7
Content-Type: text/html; charset=utf-8
```

Additionally, if we try to run the `DELETE` again (using the same original command), we get the "Gone" response again:

```
HTTP/1.0 404 GONE
Date: Fri, 20 May 2011 07:30:00 GMT
Server: WSGIServer/0.1 Python/2.7
Content-Type: text/html; charset=utf-8
```

Deleting A Whole Collection Of Resources

Finally, it's possible to remove an entire collection of resources. This is as destructive as it sounds. Once again, we use the `DELETE` method, this time on the entire list endpoint:

```
curl --dump-header - -H "Content-Type: application/json" -X DELETE  http://
↪localhost:8000/api/v1/entry/
```

As a response, we get:

```
HTTP/1.0 204 NO CONTENT
Date: Fri, 20 May 2011 07:32:51 GMT
Server: WSGIServer/0.1 Python/2.7
Content-Length: 0
Content-Type: text/html; charset=utf-8
```

Hitting the list view:

```
curl --dump-header - http://localhost:8000/api/v1/entry/
```

Gives us a 200 but no objects:

```
{
    "meta": {
        "limit": 20,
        "next": null,
        "offset": 0,
        "previous": null,
        "total_count": 0
    },
    "objects": []
}
```

Bulk Operations

As an optimization, it is possible to do many creations, updates, and deletions to a collection in a single request by sending a `PATCH` to the list endpoint.:

```
curl --dump-header - -H "Content-Type: application/json" -X PATCH --data '{"objects":␣
↪[{"body": "Surprise! Another post!.", "pub_date": "2012-02-16T00:46:38", "slug":
↪"yet-another-post", "title": "Yet Another Post"}], "deleted_objects": ["http://
↪localhost:8000/api/v1/entry/4/"]}'  http://localhost:8000/api/v1/entry/
```

We should get back:

```
HTTP/1.0 202 ACCEPTED
Date: Fri, 16 Feb 2012 00:46:38 GMT
Server: WSGIServer/0.1 Python/2.7
Content-Length: 0
Content-Type: text/html; charset=utf-8
```

The Accepted response means the server has accepted the request, but gives no details on the result. In order to see any created resources, we would need to do a get `GET` on the list endpoint.

For detailed information on the format of a bulk request, see *patch_list*.

You Did It!

That's a whirlwind tour of interacting with a Tastypie API. There's additional functionality present, such as:

- `POST`/`PUT` the other supported content-types
- More filtering/`order_by`/`limit`/`offset` tricks
- Using overridden URLconfs to support complex or non-PK lookups
- Authentication

But this grounds you in the basics & hopefully clarifies usage/debugging better.

CHAPTER 3

Tastypie Settings

This is a comprehensive list of the settings Tastypie recognizes.

API_LIMIT_PER_PAGE

Optional

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

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
```

If you don't want to limit the number of records by default, you can set this setting to 0:

```
API_LIMIT_PER_PAGE = 0
```

Defaults to 20.

TASTYPIE_FULL_DEBUG

Optional

This setting controls what the behavior is when an unhandled exception occurs.

If set to `True` and `settings.DEBUG = True`, the standard Django technical 500 is displayed.

If not set or set to `False`, Tastypie will return a serialized response. If `settings.DEBUG` is `True`, you'll get the actual exception message plus a traceback. If `settings.DEBUG` is `False`, Tastypie will call `mail_admins()` and provide a canned error message (which you can override with `TASTYPIE_CANNED_ERROR`) in the response.

An example:

```
TASTYPIE_FULL_DEBUG = True
```

Defaults to `False`.

TASTYPIE_CANNED_ERROR

Optional

This setting allows you to override the canned error response when an unhandled exception is raised and `settings.DEBUG` is `False`.

An example:

```
TASTYPIE_CANNED_ERROR = "Oops, we broke it!"
```

Defaults to `"Sorry, this request could not be processed. Please try again later."`.

TASTYPIE_ALLOW_MISSING_SLASH

Optional

This setting allows your URLs to be missing the final slash. Useful for integrating with other systems.

You must also have `settings.APPEND_SLASH = False` so that Django does not emit HTTP 302 redirects.

An example:

```
TASTYPIE_ALLOW_MISSING_SLASH = True
```

Defaults to `False`.

TASTYPIE_DATETIME_FORMATTING

Optional

This setting allows you to globally choose what format your datetime/date/time data will be formatted in. Valid options are `iso-8601`, `iso-8601-strict` & `rfc-2822`.

An example:

```
TASTYPIE_DATETIME_FORMATTING = 'rfc-2822'
```

Defaults to `iso-8601`. `iso-8601` includes microseconds if available, use `iso-8601-strict` to strip them.

TASTYPIE_DEFAULT_FORMATS

Optional

This setting allows you to globally configure the list of allowed serialization formats for your entire site.

An example:

```
TASTYPIE_DEFAULT_FORMATS = ['json', 'xml']
```

Defaults to `['json', 'xml', 'yaml', 'plist']`.

TASTYPIE_ABSTRACT_APIKEY

Optional

This setting makes the `ApiKey` model an abstract base class. This may be useful in multi-database setups where many databases each have their own table for user data and `ApiKeyAuthentication` is not used. Without this setting, the `tastypie_apikey` table would have to be created on each database containing user account data (such as Django's built-in `auth_user` table generated by `django.contrib.auth.models.User`). Valid options are `True` & `False`.

An example:

```
TASTYPIE_ABSTRACT_APIKEY = True
```

Defaults to `False`.

CHAPTER 4

Using Tastypie With Non-ORM Data Sources

Much of this documentation demonstrates the use of Tastypie with Django's ORM. You might think that Tastypie depended on the ORM, when in fact, it was purpose-built to handle non-ORM data. This documentation should help you get started providing APIs using other data sources.

Virtually all of the code that makes Tastypie actually process requests & return data is within the `Resource` class. `ModelResource` is actually a light wrapper around `Resource` that provides ORM-specific access. The methods that `ModelResource` overrides are the same ones you'll need to override when hooking up your data source.

Approach

When working with `Resource`, many things are handled for you. All the authentication/authorization/caching/serialization/throttling bits should work as normal and Tastypie can support all the REST-style methods. Schemas & discovery views all work the same as well.

What you don't get out of the box are the fields you're choosing to expose & the lowest level data access methods. If you want a full read-write API, there are nine methods you need to implement. They are:

- `detail_uri_kwargs`
- `get_object_list`
- `obj_get_list`
- `obj_get`
- `obj_create`
- `obj_update`
- `obj_delete_list`
- `obj_delete`
- `rollback`

If read-only is all you're exposing, you can cut that down to four methods to override.

Using Riak for MessageResource

As an example, we'll take integrating with Riak (a Dynamo-like NoSQL store) since it has both a simple API and demonstrate what hooking up to a non-relational datastore looks like:

```
import riak

from tastypie import fields
from tastypie.authorization import Authorization
from tastypie.resources import Resource

# We need a generic object to shove data in/get data from.
# Riak generally just tosses around dictionaries, so we'll lightly
# wrap that.
class RiakObject(object):
    def __init__(self, initial=None):
        self.__dict__['_data'] = {}

        if hasattr(initial, 'items'):
            self.__dict__['_data'] = initial

    def __getattr__(self, name):
        return self._data.get(name, None)

    def __setattr__(self, name, value):
        self.__dict__['_data'][name] = value

    def to_dict(self):
        return self._data


class MessageResource(Resource):
    # Just like a Django ``Form`` or ``Model``, we're defining all the
    # fields we're going to handle with the API here.
    uuid = fields.CharField(attribute='uuid')
    user_uuid = fields.CharField(attribute='user_uuid')
    message = fields.CharField(attribute='message')
    created = fields.IntegerField(attribute='created')

    class Meta:
        resource_name = 'riak'
        object_class = RiakObject
        authorization = Authorization()

    # Specific to this resource, just to get the needed Riak bits.
    def _client(self):
        return riak.RiakClient()

    def _bucket(self):
        client = self._client()
        # Note that we're hard-coding the bucket to use. Fine for
        # example purposes, but you'll want to abstract this.
        return client.bucket('messages')

    # The following methods will need overriding regardless of your
    # data source.
    def detail_uri_kwargs(self, bundle_or_obj):
        kwargs = {}
```

```
        if isinstance(bundle_or_obj, Bundle):
            kwargs['pk'] = bundle_or_obj.obj.uuid
        else:
            kwargs['pk'] = bundle_or_obj.uuid

        return kwargs

    def get_object_list(self, request):
        query = self._client().add('messages')
        query.map("function(v) { var data = JSON.parse(v.values[0].data); return [[v.
↪key, data]]; }")
        results = []

        for result in query.run():
            new_obj = RiakObject(initial=result[1])
            new_obj.uuid = result[0]
            results.append(new_obj)

        return results

    def obj_get_list(self, bundle, **kwargs):
        # Filtering disabled for brevity...
        return self.get_object_list(bundle.request)

    def obj_get(self, bundle, **kwargs):
        bucket = self._bucket()
        message = bucket.get(kwargs['pk'])
        return RiakObject(initial=message.get_data())

    def obj_create(self, bundle, **kwargs):
        bundle.obj = RiakObject(initial=kwargs)
        bundle = self.full_hydrate(bundle)
        bucket = self._bucket()
        new_message = bucket.new(bundle.obj.uuid, data=bundle.obj.to_dict())
        new_message.store()
        return bundle

    def obj_update(self, bundle, **kwargs):
        return self.obj_create(bundle, **kwargs)

    def obj_delete_list(self, bundle, **kwargs):
        bucket = self._bucket()

        for key in bucket.get_keys():
            obj = bucket.get(key)
            obj.delete()

    def obj_delete(self, bundle, **kwargs):
        bucket = self._bucket()
        obj = bucket.get(kwargs['pk'])
        obj.delete()

    def rollback(self, bundles):
        pass
```

This represents a full, working, Riak-powered API endpoint. All REST-style actions (GET/POST/PUT/DELETE) work correctly. The only shortcut taken in this example was skipping filter-abilty, as adding in the MapReduce bits

would have decreased readability.

All said and done, just nine methods needed overriding, eight of which were highly specific to how data access is done.

CHAPTER 5

Tools

Here are some tools that might help in interacting with the API that Tastypie provides:

Browser

JSONView

- Firefox - https://addons.mozilla.org/en-US/firefox/addon/jsonview/
- Chrome - https://chrome.google.com/webstore/detail/chklaanhfefbnpoihckbnefhakgolnmc

A plugin (actually two different ones that closely mirror each other) that nicely reformats JSON data in the browser.

Postman - Rest Client

- Chrome - https://chrome.google.com/webstore/detail/fdmmgilgnpjigdojojpjoooidkmcomcm

A feature rich Chrome extension with JSON and XML support

Extensions

Tastypie-msgpack

https://github.com/stephenmcd/tastypie-msgpack

Adds MsgPack support to Tastypie's serializer.

Python

Slumber

http://slumber.in/

Slumber is a small Python library that makes it easy to access & work with APIs. It works for many others, but works especially well with Tastypie.

Hammock

https://github.com/kadirpekel/hammock

Hammock is a fun module lets you deal with rest APIs by converting them into dead simple programmatic APIs. It uses popular `requests` module in backyard to provide full-fledged rest experience.

Here is what it looks like:

```
>>> import hammock
>>> api = hammock.Hammock('http://localhost:8000')
>>> api.users('foo').posts('bar').comments.GET()
<Response [200]>
```

drest

https://drest.readthedocs.io/

drest is another small Python library. It focuses on extensibility & can also work with many different API, with an emphasis on Tastypie.

httpie

https://github.com/jkbr/httpie

HTTPie is a command line HTTP client written in Python. Its goal is to make command-line interaction with web services as human-friendly as possible and allows much conciser statements compared with curl.

For example for POSTing a JSON object you simply call:

> $ http localhost:8000/api/v1/entry/ title="Foo" body="Bar" user="/api/v1/user/1/"

Now compare this with curl:

> $ curl –dump-header - -H "Content-Type: application/json" -X POST –data '{"title": "Foo", "body": "Bar", "user": "/api/v1/user/1/"}' http://localhost:8000/api/v1/entry/

json.tool

Included with Python, this tool makes reformatting JSON easy. For example:

```
$ curl http://localhost:8000/api/v1/note/ | python -m json.tool
```

Will return nicely reformatted data like:

```
{
    "meta": {
        "total_count": 1
    },
    "objects": [
        {
            "content": "Hello world!",
            "user": "/api/v1/user/1/"
        }
    ]
}
```

django-permissionsx

https://github.com/thinkingpotato/django-permissionsx

This package allows using one set of rules both for Django class-based views] and Tastypie authorization. For example:

articles/permissions.py:

```
class StaffPermissions(Permissions):
    permissions = P(profile__is_editor=True) | P(profile__is_administrator=True)
```

articles/views.py:

```
class ArticleDeleteView(PermissionsViewMixin, DeleteView):
    model = Article
    success_url = reverse_lazy('article_list')
    permissions = StaffPermissions
```

articles/api.py:

```
class StaffOnlyAuthorization(TastypieAuthorization):
    permissions_class = StaffPermissions
```

django-superbulk

https://github.com/thelonecabbage/django-superbulk

This app adds bulk operation support to any Django view-based app, allowing for better transactional behavior.

Javascript

backbone-tastypie

https://github.com/PaulUithol/backbone-tastypie

A small layer that makes Backbone & Tastypie plan nicely together.

backbone-relational

https://github.com/PaulUithol/Backbone-relational/

Allows Backbone to work with relational data, like the kind of data Tastypie provides.

CHAPTER 6

Testing

Having integrated unit tests that cover your API's behavior is important, as it helps provide verification that your API code is still valid & working correctly with the rest of your application.

Tastypie provides some basic facilities that build on top of Django's testing support, in the form of a specialized `TestApiClient` & `ResourceTestCaseMixin`.

The `ResourceTestCaseMixin` can be used along with Django's `TestCase` or other Django test classes. It provides quite a few extra assertion methods that are specific to APIs. Under the hood, it uses the `TestApiClient` to perform requests properly.

The `TestApiClient` builds on & exposes an interface similar to that of Django's `Client`. However, under the hood, it hands all the setup needed to construct a proper request.

Example Usage

The typical use case will primarily consist of adding the `ResourceTestCaseMixin` class to an ordinary Django test class & using the built-in assertions to ensure your API is behaving correctly. For the purposes of this example, we'll assume the resource in question looks like:

```
from tastypie.authentication import BasicAuthentication
from tastypie.resources import ModelResource
from entries.models import Entry


class EntryResource(ModelResource):
    class Meta:
        queryset = Entry.objects.all()
        authentication = BasicAuthentication()
```

An example usage might look like:

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

```
from django.test import TestCase
from tastypie.test import ResourceTestCaseMixin
from entries.models import Entry


class EntryResourceTest(ResourceTestCaseMixin, TestCase):
    # Use ``fixtures`` & ``urls`` as normal. See Django's ``TestCase``
    # documentation for the gory details.
    fixtures = ['test_entries.json']

    def setUp(self):
        super(EntryResourceTest, self).setUp()

        # Create a user.
        self.username = 'daniel'
        self.password = 'pass'
        self.user = User.objects.create_user(self.username, 'daniel@example.com',␣
↪self.password)

        # Fetch the ``Entry`` object we'll use in testing.
        # Note that we aren't using PKs because they can change depending
        # on what other tests are running.
        self.entry_1 = Entry.objects.get(slug='first-post')

        # We also build a detail URI, since we will be using it all over.
        # DRY, baby. DRY.
        self.detail_url = '/api/v1/entry/{0}/'.format(self.entry_1.pk)

        # The data we'll send on POST requests. Again, because we'll use it
        # frequently (enough).
        self.post_data = {
            'user': '/api/v1/user/{0}/'.format(self.user.pk),
            'title': 'Second Post!',
            'slug': 'second-post',
            'created': '2012-05-01T22:05:12'
        }

    def get_credentials(self):
        return self.create_basic(username=self.username, password=self.password)

    def test_get_list_unauthenticated(self):
        self.assertHttpUnauthorized(self.api_client.get('/api/v1/entries/', format=
↪'json'))

    def test_get_list_json(self):
        resp = self.api_client.get('/api/v1/entries/', format='json',␣
↪authentication=self.get_credentials())
        self.assertValidJSONResponse(resp)

        # Scope out the data for correctness.
        self.assertEqual(len(self.deserialize(resp)['objects']), 12)
        # Here, we're checking an entire structure for the expected data.
        self.assertEqual(self.deserialize(resp)['objects'][0], {
            'pk': str(self.entry_1.pk),
            'user': '/api/v1/user/{0}/'.format(self.user.pk),
            'title': 'First post',
            'slug': 'first-post',
            'created': '2012-05-01T19:13:42',
```

```
            'resource_uri': '/api/v1/entry/{0}/'.format(self.entry_1.pk)
        })

    def test_get_list_xml(self):
        self.assertValidXMLResponse(self.api_client.get('/api/v1/entries/', format=
↪'xml', authentication=self.get_credentials()))

    def test_get_detail_unauthenticated(self):
        self.assertHttpUnauthorized(self.api_client.get(self.detail_url, format='json
↪'))

    def test_get_detail_json(self):
        resp = self.api_client.get(self.detail_url, format='json',␣
↪authentication=self.get_credentials())
        self.assertValidJSONResponse(resp)

        # We use ``assertKeys`` here to just verify the keys, not all the data.
        self.assertKeys(self.deserialize(resp), ['created', 'slug', 'title', 'user'])
        self.assertEqual(self.deserialize(resp)['name'], 'First post')

    def test_get_detail_xml(self):
        self.assertValidXMLResponse(self.api_client.get(self.detail_url, format='xml',
↪ authentication=self.get_credentials()))

    def test_post_list_unauthenticated(self):
        self.assertHttpUnauthorized(self.api_client.post('/api/v1/entries/', format=
↪'json', data=self.post_data))

    def test_post_list(self):
        # Check how many are there first.
        self.assertEqual(Entry.objects.count(), 5)
        self.assertHttpCreated(self.api_client.post('/api/v1/entries/', format='json',
↪ data=self.post_data, authentication=self.get_credentials()))
        # Verify a new one has been added.
        self.assertEqual(Entry.objects.count(), 6)

    def test_put_detail_unauthenticated(self):
        self.assertHttpUnauthorized(self.api_client.put(self.detail_url, format='json
↪', data={}))

    def test_put_detail(self):
        # Grab the current data & modify it slightly.
        original_data = self.deserialize(self.api_client.get(self.detail_url, format=
↪'json', authentication=self.get_credentials()))
        new_data = original_data.copy()
        new_data['title'] = 'Updated: First Post'
        new_data['created'] = '2012-05-01T20:06:12'

        self.assertEqual(Entry.objects.count(), 5)
        self.assertHttpAccepted(self.api_client.put(self.detail_url, format='json',␣
↪data=new_data, authentication=self.get_credentials()))
        # Make sure the count hasn't changed & we did an update.
        self.assertEqual(Entry.objects.count(), 5)
        # Check for updated data.
        self.assertEqual(Entry.objects.get(pk=25).title, 'Updated: First Post')
        self.assertEqual(Entry.objects.get(pk=25).slug, 'first-post')
        self.assertEqual(Entry.objects.get(pk=25).created, datetime.datetime(2012, 3,␣
↪1, 13, 6, 12))
```

```
    def test_delete_detail_unauthenticated(self):
        self.assertHttpUnauthorized(self.api_client.delete(self.detail_url, format=
↪'json'))

    def test_delete_detail(self):
        self.assertEqual(Entry.objects.count(), 5)
        self.assertHttpAccepted(self.api_client.delete(self.detail_url, format='json',
↪ authentication=self.get_credentials()))
        self.assertEqual(Entry.objects.count(), 4)
```

Note that this example doesn't cover other cases, such as filtering, `PUT` to a list endpoint, `DELETE` to a list endpoint, `PATCH` support, etc.

`ResourceTestCaseMixin` API Reference

The `ResourceTestCaseMixin` exposes the following methods for use. Most are enhanced assertions or provide API-specific behaviors.

`get_credentials`

`ResourceTestCaseMixin.`**`get_credentials`**(*self*)

A convenience method for the user as a way to shorten up the often repetitious calls to create the same authentication.

Raises `NotImplementedError` by default.

Usage:

```
class MyResourceTestCase(ResourceTestCaseMixin, TestCase):
    def get_credentials(self):
        return self.create_basic('daniel', 'pass')

    # Then the usual tests...
```

`create_basic`

`ResourceTestCaseMixin.`**`create_basic`**(*self*, *username*, *password*)

Creates & returns the HTTP `Authorization` header for use with BASIC Auth.

`create_apikey`

`ResourceTestCaseMixin.`**`create_apikey`**(*self*, *username*, *api_key*)

Creates & returns the HTTP `Authorization` header for use with `ApiKeyAuthentication`.

`create_digest`

`ResourceTestCaseMixin.`**`create_digest`**(*self*, *username*, *api_key*, *method*, *uri*)

Creates & returns the HTTP `Authorization` header for use with Digest Auth.

create_oauth

ResourceTestCaseMixin.**create_oauth**(*self*, *user*)

Creates & returns the HTTP `Authorization` header for use with Oauth.

assertHttpOK

ResourceTestCaseMixin.**assertHttpOK**(*self*, *resp*)

Ensures the response is returning a HTTP 200.

assertHttpCreated

ResourceTestCaseMixin.**assertHttpCreated**(*self*, *resp*)

Ensures the response is returning a HTTP 201.

assertHttpAccepted

ResourceTestCaseMixin.**assertHttpAccepted**(*self*, *resp*)

Ensures the response is returning either a HTTP 202 or a HTTP 204.

assertHttpMultipleChoices

ResourceTestCaseMixin.**assertHttpMultipleChoices**(*self*, *resp*)

Ensures the response is returning a HTTP 300.

assertHttpSeeOther

ResourceTestCaseMixin.**assertHttpSeeOther**(*self*, *resp*)

Ensures the response is returning a HTTP 303.

assertHttpNotModified

ResourceTestCaseMixin.**assertHttpNotModified**(*self*, *resp*)

Ensures the response is returning a HTTP 304.

assertHttpBadRequest

ResourceTestCaseMixin.**assertHttpBadRequest**(*self*, *resp*)

Ensures the response is returning a HTTP 400.

assertHttpUnauthorized

ResourceTestCaseMixin.**assertHttpUnauthorized**(*self*, *resp*)

Ensures the response is returning a HTTP 401.

assertHttpForbidden

ResourceTestCaseMixin.**assertHttpForbidden**(*self*, *resp*)

Ensures the response is returning a HTTP 403.

assertHttpNotFound

ResourceTestCaseMixin.**assertHttpNotFound**(*self*, *resp*)

Ensures the response is returning a HTTP 404.

assertHttpMethodNotAllowed

ResourceTestCaseMixin.**assertHttpMethodNotAllowed**(*self*, *resp*)

Ensures the response is returning a HTTP 405.

assertHttpConflict

ResourceTestCaseMixin.**assertHttpConflict**(*self*, *resp*)

Ensures the response is returning a HTTP 409.

assertHttpGone

ResourceTestCaseMixin.**assertHttpGone**(*self*, *resp*)

Ensures the response is returning a HTTP 410.

assertHttpTooManyRequests

ResourceTestCaseMixin.**assertHttpTooManyRequests**(*self*, *resp*)

Ensures the response is returning a HTTP 429.

assertHttpApplicationError

ResourceTestCaseMixin.**assertHttpApplicationError**(*self*, *resp*)

Ensures the response is returning a HTTP 500.

assertHttpNotImplemented

ResourceTestCaseMixin.**assertHttpNotImplemented**(*self*, *resp*)

Ensures the response is returning a HTTP 501.

assertValidJSON

ResourceTestCaseMixin.**assertValidJSON**(*self*, *data*)

Given the provided `data` as a string, ensures that it is valid JSON & can be loaded properly.

assertValidXML

ResourceTestCaseMixin.**assertValidXML**(*self*, *data*)

Given the provided `data` as a string, ensures that it is valid XML & can be loaded properly.

assertValidYAML

ResourceTestCaseMixin.**assertValidYAML**(*self*, *data*)

Given the provided `data` as a string, ensures that it is valid YAML & can be loaded properly.

assertValidPlist

ResourceTestCaseMixin.**assertValidPlist**(*self*, *data*)

Given the provided `data` as a string, ensures that it is valid binary plist & can be loaded properly.

assertValidJSONResponse

ResourceTestCaseMixin.**assertValidJSONResponse**(*self*, *resp*)

Given a `HttpResponse` coming back from using the `client`, assert that you get back:

- An HTTP 200
- The correct content-type (`application/json`)
- The content is valid JSON

assertValidXMLResponse

ResourceTestCaseMixin.**assertValidXMLResponse**(*self*, *resp*)

Given a `HttpResponse` coming back from using the `client`, assert that you get back:

- An HTTP 200
- The correct content-type (`application/xml`)
- The content is valid XML

assertValidYAMLResponse

ResourceTestCaseMixin.**assertValidYAMLResponse**(*self*, *resp*)

Given a `HttpResponse` coming back from using the `client`, assert that you get back:

- An HTTP 200
- The correct content-type (`text/yaml`)
- The content is valid YAML

assertValidPlistResponse

ResourceTestCaseMixin.**assertValidPlistResponse**(*self*, *resp*)

Given a HttpResponse coming back from using the client, assert that you get back:

- An HTTP 200
- The correct content-type (application/x-plist)
- The content is valid binary plist data

deserialize

ResourceTestCaseMixin.**deserialize**(*self*, *resp*)

Given a HttpResponse coming back from using the client, this method checks the Content-Type header & attempts to deserialize the data based on that.

It returns a Python datastructure (typically a dict) of the serialized data.

serialize

ResourceTestCaseMixin.**serialize**(*self*, *data*, *format='application/json'*)

Given a Python datastructure (typically a dict) & a desired content-type, this method will return a serialized string of that data.

assertKeys

ResourceTestCaseMixin.**assertKeys**(*self*, *data*, *expected*)

This method ensures that the keys of the data match up to the keys of expected.

It covers the (extremely) common case where you want to make sure the keys of a response match up to what is expected. This is typically less fragile than testing the full structure, which can be prone to data changes.

ResourceTestCase API Reference

ResourceTestCase is deprecated and will be removed by v1.0.0.

class MyTest(ResourceTestCase) is equivalent to class MyTest(ResourceTestCaseMixin, TestCase).

TestApiClient API Reference

The TestApiClient simulates a HTTP client making calls to the API. It's important to note that it uses Django's testing infrastructure, so it's not making actual calls against a webserver.

__init__

TestApiClient.**__init__**(*self*, *serializer=None*)

Sets up a fresh TestApiClient instance.

If you are employing a custom serializer, you can pass the class to the serializer= kwarg.

get_content_type

TestApiClient.**get_content_type**(*self*, *short_format*)

Given a short name (such as `json` or `xml`), returns the full content-type for it (`application/json` or `application/xml` in this case).

get

TestApiClient.**get**(*self*, *uri*, *format='json'*, *data=None*, *authentication=None*, ***kwargs*)

Performs a simulated `GET` request to the provided URI.

Optionally accepts a `data` kwarg, which in the case of `GET`, lets you send along `GET` parameters. This is useful when testing filtering or other things that read off the `GET` params. Example:

```
from tastypie.test import TestApiClient
client = TestApiClient()

response = client.get('/api/v1/entry/1/', data={'format': 'json', 'title__startswith
↪': 'a', 'limit': 20, 'offset': 60})
```

Optionally accepts an `authentication` kwarg, which should be an HTTP header with the correct authentication data already setup.

All other `**kwargs` passed in get passed through to the Django `TestClient`. See https://docs.djangoproject.com/en/dev/topics/testing/#module-django.test.client for details.

post

TestApiClient.**post**(*self*, *uri*, *format='json'*, *data=None*, *authentication=None*, ***kwargs*)

Performs a simulated `POST` request to the provided URI.

Optionally accepts a `data` kwarg. **Unlike** `GET`, in `POST` the `data` gets serialized & sent as the body instead of becoming part of the URI. Example:

```
from tastypie.test import TestApiClient
client = TestApiClient()

response = client.post('/api/v1/entry/', data={
    'created': '2012-05-01T20:02:36',
    'slug': 'another-post',
    'title': 'Another Post',
    'user': '/api/v1/user/1/',
})
```

Optionally accepts an `authentication` kwarg, which should be an HTTP header with the correct authentication data already setup.

All other `**kwargs` passed in get passed through to the Django `TestClient`. See https://docs.djangoproject.com/en/dev/topics/testing/#module-django.test.client for details.

put

TestApiClient.**put**(*self*, *uri*, *format='json'*, *data=None*, *authentication=None*, ***kwargs*)

Performs a simulated `PUT` request to the provided URI.

Optionally accepts a `data` kwarg. **Unlike** `GET`, in `PUT` the `data` gets serialized & sent as the body instead of becoming part of the URI. Example:

```
from tastypie.test import TestApiClient
client = TestApiClient()

response = client.put('/api/v1/entry/1/', data={
    'created': '2012-05-01T20:02:36',
    'slug': 'another-post',
    'title': 'Another Post',
    'user': '/api/v1/user/1/',
})
```

Optionally accepts an `authentication` kwarg, which should be an HTTP header with the correct authentication data already setup.

All other `**kwargs` passed in get passed through to the Django `TestClient`. See https://docs.djangoproject.com/en/dev/topics/testing/#module-django.test.client for details.

patch

`TestApiClient.`**`patch`**(*self*, *uri*, *format='json'*, *data=None*, *authentication=None*, ***kwargs*)

Performs a simulated `PATCH` request to the provided URI.

Optionally accepts a `data` kwarg. **Unlike** `GET`, in `PATCH` the `data` gets serialized & sent as the body instead of becoming part of the URI. Example:

```
from tastypie.test import TestApiClient
client = TestApiClient()

response = client.patch('/api/v1/entry/1/', data={
    'created': '2012-05-01T20:02:36',
    'slug': 'another-post',
    'title': 'Another Post',
    'user': '/api/v1/user/1/',
})
```

Optionally accepts an `authentication` kwarg, which should be an HTTP header with the correct authentication data already setup.

All other `**kwargs` passed in get passed through to the Django `TestClient`. See https://docs.djangoproject.com/en/dev/topics/testing/#module-django.test.client for details.

delete

`TestApiClient.`**`delete`**(*self*, *uri*, *format='json'*, *data=None*, *authentication=None*, ***kwargs*)

Performs a simulated `DELETE` request to the provided URI.

Optionally accepts a `data` kwarg, which in the case of `DELETE`, lets you send along `DELETE` parameters. This is useful when testing filtering or other things that read off the `DELETE` params. Example:

```
from tastypie.test import TestApiClient
client = TestApiClient()
```

```
response = client.delete('/api/v1/entry/1/', data={'format': 'json'})
```

Optionally accepts an `authentication` kwarg, which should be an HTTP header with the correct authentication data already setup.

All other `**kwargs` passed in get passed through to the Django `TestClient`. See https://docs.djangoproject.com/en/dev/topics/testing/#module-django.test.client for details.

CHAPTER 7

Compatibility Notes

Tastypie does its best to be a good third-party app, trying to interoperate with the widest range of Django environments it can. However, there are times where certain things aren't possible. We'll do our best to document them here.

`ApiKey` Database Index

When the `ApiKey` model was added to Tastypie, an index was lacking on the `key` field. This was the case until the v0.9.12 release. The model was updated & a migration was added (`0002_add_apikey_index.py`). However, due to the way MySQL works & the way Django generates index names, this migration would fail miserably on many MySQL installs.

If you are using MySQL & the `ApiKey` authentication class, you may need to manually add an index for the the `ApiKey.key` field. Something to the effect of:

```
BEGIN; -- LOLMySQL
CREATE INDEX tastypie_apikey_key_index ON tastypie_apikey (`key`);
COMMIT;
```

CHAPTER 8

Python 3 Support

As of Tastypie v0.10.0, it has been ported to support both Python 2 & Python 3 within the same codebase. This builds on top of what six & Django provide.

No changes are required for anyone running an existing Tastypie installation. The API is completely backward-compatible, so you should be able to run your existing software without modification.

All tests pass under both Python 2 & 3.

Incompatibilities

Oauth Is Unsupported

Tastypie was depending on several Oauth libraries for that authentication mechanism. Unfortunately, none of them have been ported to Python 3. They're still usable from Python 2, but that will be blocked until the underlying libraries port (or an alternative can be found).

Changed Requirements

Several requirements have changed under Python 3 (mostly due to unofficial ports). They are:

- python3-digest instead of python-digest
- python-mimeparse instead of mimeparse

Notes

Request/Response Bodies

For explicitness, Django on Python 3 reads request bodies & sends response bodies as **binary** data. This requires an explicit `.decode('utf-8')` that was not required (but works fine) under Python 2. If you're sending or reading

the bodies from Python, you'll need to keep this in mind.

Testing

If you were testing things such as the XML/JSON generated by a given response, under Python 3.3.2+, hash randomization is in effect, which means that the ordering of dictionaries is no longer consistent, even on the same platform.

To mitigate this, Tastypie now tries to ensure that serialized data is sorted alphabetically. So if you were making string assertions, you'll need to update them for the new payloads.

CHAPTER 9

Resources

In terms of a REST-style architecture, a "resource" is a collection of similar data. This data could be a table of a database, a collection of other resources or a similar form of data storage. In Tastypie, these resources are generally intermediaries between the end user & objects, usually Django models. As such, `Resource` (and its model-specific twin `ModelResource`) form the heart of Tastypie's functionality.

Quick Start

A sample resource definition might look something like:

```
from django.contrib.auth.models import User
from tastypie import fields
from tastypie.authorization import DjangoAuthorization
from tastypie.resources import ModelResource, ALL, ALL_WITH_RELATIONS
from myapp.models import Entry


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


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

    class Meta:
        queryset = Entry.objects.all()
        list_allowed_methods = ['get', 'post']
        detail_allowed_methods = ['get', 'post', 'put', 'delete']
        resource_name = 'myapp/entry'
        authorization = DjangoAuthorization()
```

```
        filtering = {
            'slug': ALL,
            'user': ALL_WITH_RELATIONS,
            'created': ['exact', 'range', 'gt', 'gte', 'lt', 'lte'],
        }
```

Why Class-Based?

Using class-based resources make it easier to extend/modify the code to meet your needs. APIs are rarely a one-size-fits-all problem space, so Tastypie tries to get the fundamentals right and provide you with enough hooks to customize things to work your way.

As is standard, this raises potential problems for thread-safety. Tastypie has been designed to minimize the possibility of data "leaking" between threads. This does however sometimes introduce some small complexities & you should be careful not to store state on the instances if you're going to be using the code in a threaded environment.

Why `Resource` vs. `ModelResource`?

Make no mistake that Django models are far and away the most popular source of data. However, in practice, there are many times where the ORM isn't the data source. Hooking up things like a NoSQL store (see *Using Tastypie With Non-ORM Data Sources*), a search solution like Haystack or even managed filesystem data are all good use cases for `Resource` knowing nothing about the ORM.

Flow Through The Request/Response Cycle

Tastypie can be thought of as a set of class-based views that provide the API functionality. As such, many part of the request/response cycle are standard Django behaviors. For instance, all routing/middleware/response-handling aspects are the same as a typical Django app. Where it differs is in the view itself.

As an example, we'll walk through what a GET request to a list endpoint (say `/api/v1/user/?format=json`) looks like:

- The `Resource.urls` are checked by Django's url resolvers.
- On a match for the list view, `Resource.wrap_view('dispatch_list')` is called. `wrap_view` provides basic error handling & allows for returning serialized errors.
- Because `dispatch_list` was passed to `wrap_view`, `Resource.dispatch_list` is called next. This is a thin wrapper around `Resource.dispatch`.
- `dispatch` does a bunch of heavy lifting. It ensures:
  - the requested HTTP method is in `allowed_methods` (`method_check`),
  - the class has a method that can handle the request (`get_list`),
  - the user is authenticated (`is_authenticated`),
  - & the user has not exceeded their throttle (`throttle_check`).

  At this point, `dispatch` actually calls the requested method (`get_list`).
- `get_list` does the actual work of the API. It does:

  - A fetch of the available objects via `Resource.obj_get_list`. In the case of `ModelResource`, this builds the ORM filters to apply (`ModelResource.build_filters`). It then gets the `QuerySet` via `ModelResource.get_object_list` (which performs `Resource.authorized_read_list` to possibly limit the set the user can work with) and applies the built filters to it.
  - It then sorts the objects based on user input (`ModelResource.apply_sorting`).
  - Then it paginates the results using the supplied `Paginator` & pulls out the data to be serialized.
  - The objects in the page have `full_dehydrate` applied to each of them, causing Tastypie to translate the raw object data into the fields the endpoint supports.
  - Finally, it calls `Resource.create_response`.
- `create_response` is a shortcut method that:
  - Determines the desired response format (`Resource.determine_format`),
  - Serializes the data given to it in the proper format,
  - And returns a Django `HttpResponse` (200 OK) with the serialized data.
- We bubble back up the call stack to `dispatch`. The last thing `dispatch` does is potentially store that a request occurred for future throttling (`Resource.log_throttled_access`) then either returns the `HttpResponse` or wraps whatever data came back in a response (so Django doesn't freak out).

Processing on other endpoints or using the other HTTP methods results in a similar cycle, usually differing only in what "actual work" method gets called (which follows the format of "`<http_method>_<list_or_detail>`"). In the case of POST/PUT, the `hydrate` cycle additionally takes place and is used to take the user data & convert it to raw data for storage.

Why Resource URIs?

Resource URIs play a heavy role in how Tastypie delivers data. This can seem very different from other solutions which simply inline related data. Though Tastypie can inline data like that (using `full=True` on the field with the relation), the default is to provide URIs.

URIs are useful because it results in smaller payloads, letting you fetch only the data that is important to you. You can imagine an instance where an object has thousands of related items that you may not be interested in.

URIs are also very cache-able, because the data at each endpoint is less likely to frequently change.

And URIs encourage proper use of each endpoint to display the data that endpoint covers.

Ideology aside, you should use whatever suits you. If you prefer fewer requests & fewer endpoints, use of `full=True` is available, but be aware of the consequences of each approach.

Accessing The Current Request

Being able to change behavior based on the current request is a very commmon need. Virtually anywhere within `Resource/ModelResource`, if a `bundle` is available, you can access it using `bundle.request`. This is useful for altering querysets, ensuring headers are present, etc.

Most methods you may need to override/extend should get a `bundle` passed to them.

If you're using the `Resource/ModelResource` directly, with no `request` available, an empty `Request` will be supplied instead. If this is a common pattern/usage in your code, you'll want to accommodate for data that potentially isn't there.

Advanced Data Preparation

Not all data can be easily pulled off an object/model attribute. And sometimes, you (or the client) may need to send data that doesn't neatly fit back into the data model on the server side. For this, Tastypie has the "dehydrate/hydrate" cycle.

The Dehydrate Cycle

Tastypie uses a "dehydrate" cycle to prepare data for serialization, which is to say that it takes the raw, potentially complicated data model & turns it into a (generally simpler) processed data structure for client consumption. This usually means taking a complex data object & turning it into a dictionary of simple data types.

Broadly speaking, this takes the `bundle.obj` instance & builds `bundle.data`, which is what is actually serialized.

The cycle looks like:

- Put the data model into a `Bundle` instance, which is then passed through the various methods.
- Run through all fields on the `Resource`, letting each field perform its own `dehydrate` method on the `bundle`.
- While processing each field, look for a `dehydrate_<fieldname>` method on the `Resource`. If it's present, call it with the `bundle`.
- Finally, after all fields are processed, if the `dehydrate` method is present on the `Resource`, it is called & given the entire `bundle`.

The goal of this cycle is to populate the `bundle.data` dictionary with data suitable for serialization. With the exception of the `alter_*` methods (as hooks to manipulate the overall structure), this cycle controls what is actually handed off to be serialized & sent to the client.

Per-field `dehydrate`

Each field (even custom `ApiField` subclasses) has its own `dehydrate` method. If it knows how to access data (say, given the `attribute` kwarg), it will attempt to populate values itself.

The return value is put in the `bundle.data` dictionary (by the `Resource`) with the fieldname as the key.

`dehydrate_FOO`

Since not all data may be ready for consumption based on just attribute access (or may require an advanced lookup/calculation), this hook enables you to fill in data or massage whatever the field generated.

Note: The `FOO` here is not literal. Instead, it is a placeholder that should be replaced with the fieldname in question.

Defining these methods is especially common when denormalizing related data, providing statistics or filling in unrelated data.

A simple example:

```
class MyResource(ModelResource):
    # The ``title`` field is already added to the class by ``ModelResource``
    # and populated off ``Note.title``. But we want allcaps titles...

    class Meta:
```

```
        queryset = Note.objects.all()

    def dehydrate_title(self, bundle):
        return bundle.data['title'].upper()
```

A complex example:

```
class MyResource(ModelResource):
    # As is, this is just an empty field. Without the ``dehydrate_rating``
    # method, no data would be populated for it.
    rating = fields.FloatField(readonly=True)

    class Meta:
        queryset = Note.objects.all()

    def dehydrate_rating(self, bundle):
        total_score = 0.0

        # Make sure we don't have to worry about "divide by zero" errors.
        if not bundle.obj.rating_set.count():
            return total_score

        # We'll run over all the ``Rating`` objects & calculate an average.
        for rating in bundle.obj.rating_set.all():
            total_score += rating.rating

        return total_score /  bundle.obj.rating_set.count()
```

The return value is updated in the `bundle.data`. You should avoid altering `bundle.data` here if you can help it.

dehydrate

The `dehydrate` method takes a now fully-populated `bundle.data` & make any last alterations to it. This is useful for when a piece of data might depend on more than one field, if you want to shove in extra data that isn't worth having its own field or if you want to dynamically remove things from the data to be returned.

A simple example:

```
class MyResource(ModelResource):
    class Meta:
        queryset = Note.objects.all()

    def dehydrate(self, bundle):
        # Include the request IP in the bundle.
        bundle.data['request_ip'] = bundle.request.META.get('REMOTE_ADDR')
        return bundle
```

A complex example:

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

    def dehydrate(self, bundle):
        # If they're requesting their own record, add in their email address.
        if bundle.request.user.pk == bundle.obj.pk:
```

```
            # Note that there isn't an ``email`` field on the ``Resource``.
            # By this time, it doesn't matter, as the built data will no
            # longer be checked against the fields on the ``Resource``.
            bundle.data['email'] = bundle.obj.email

        return bundle
```

This method should return a `bundle`, whether it modifies the existing one or creates a whole new one. You can even remove any/all data from the `bundle.data` if you wish.

The Hydrate Cycle

Tastypie uses a "hydrate" cycle to take serialized data from the client and turn it into something the data model can use. This is the reverse process from the `dehydrate` cycle. In fact, by default, Tastypie's serialized data should be "round-trip-able", meaning the data that comes out should be able to be fed back in & result in the same original data model. This usually means taking a dictionary of simple data types & turning it into a complex data object.

Broadly speaking, this takes the recently-deserialized `bundle.data` dictionary & builds `bundle.obj` (but does **NOT** save it).

The cycle looks like:

- Put the data from the client into a `Bundle` instance, which is then passed through the various methods.
- If the `hydrate` method is present on the `Resource`, it is called & given the entire `bundle`.
- Then run through all fields on the `Resource`, look for a `hydrate_<fieldname>` method on the `Resource`. If it's present, call it with the `bundle`.
- Finally after all other processing is done, while processing each field, let each field perform its own `hydrate` method on the `bundle`.

The goal of this cycle is to populate the `bundle.obj` data model with data suitable for saving/persistence. Again, with the exception of the `alter_*` methods (as hooks to manipulate the overall structure), this cycle controls how the data from the client is interpreted & placed on the data model.

hydrate

The `hydrate` method allows you to make initial changes to the `bundle.obj`. This includes things like prepopulating fields you don't expose over the API, recalculating related data or mangling data.

Example:

```
class MyResource(ModelResource):
    # The ``title`` field is already added to the class by ``ModelResource``
    # and populated off ``Note.title``. We'll use that title to build a
    # ``Note.slug`` as well.

    class Meta:
        queryset = Note.objects.all()

    def hydrate(self, bundle):
        # Don't change existing slugs.
        # In reality, this would be better implemented at the ``Note.save``
        # level, but is for demonstration.
        if not bundle.obj.pk:
            bundle.obj.slug = slugify(bundle.data['title'])
```

```
        return bundle
```

This method should return a `bundle`, whether it modifies the existing one or creates a whole new one. You can even remove any/all data from the `bundle.obj` if you wish.

hydrate_FOO

Data from the client may not map directly onto the data model or might need augmentation. This hook lets you take that data & convert it.

Note: The `FOO` here is not literal. Instead, it is a placeholder that should be replaced with the fieldname in question.

A simple example:

```
class MyResource(ModelResource):
    # The ``title`` field is already added to the class by ``ModelResource``
    # and populated off ``Note.title``. But we want lowercase titles...

    class Meta:
        queryset = Note.objects.all()

    def hydrate_title(self, bundle):
        bundle.data['title'] = bundle.data['title'].lower()
        return bundle
```

The return value is the `bundle`.

Per-field hydrate

Each field (even custom `ApiField` subclasses) has its own `hydrate` method. If it knows how to access data (say, given the `attribute` kwarg), it will attempt to take data from the `bundle.data` & assign it on the data model.

The return value is put in the `bundle.obj` attribute for that fieldname.

Reverse "Relationships"

Unlike Django's ORM, Tastypie does not automatically create reverse relations. This is because there is substantial technical complexity involved, as well as perhaps unintentionally exposing related data in an incorrect way to the end user of the API.

However, it is still possible to create reverse relations. Instead of handing the `ToOneField` or `ToManyField` a class, pass them a string that represents the full path to the desired class. Implementing a reverse relationship looks like so:

```
# myapp/api/resources.py
from tastypie import fields
from tastypie.resources import ModelResource
from myapp.models import Note, Comment


class NoteResource(ModelResource):
```

```
    comments = fields.ToManyField('myapp.api.resources.CommentResource', 'comments')

    class Meta:
        queryset = Note.objects.all()


class CommentResource(ModelResource):
    note = fields.ToOneField(NoteResource, 'notes')

    class Meta:
        queryset = Comment.objects.all()
```

Warning: Unlike Django, you can't use just the class name (i.e. `'CommentResource'`), even if it's in the same module. Tastypie (intentionally) lacks a construct like the `AppCache` which makes that sort of thing work in Django. Sorry.

Tastypie also supports self-referential relations. If you assume we added the appropriate self-referential `ForeignKey` to the `Note` model, implementing a similar relation in Tastypie would look like:

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


class NoteResource(ModelResource):
    sub_notes = fields.ToManyField('self', 'notes')

    class Meta:
        queryset = Note.objects.all()
```

Resource Options (AKA `Meta`)

The inner `Meta` class allows for class-level configuration of how the `Resource` should behave. The following options are available:

`serializer`

Controls which serializer class the `Resource` should use. Default is `tastypie.serializers.Serializer()`.

`authentication`

Controls which authentication class the `Resource` should use. Default is `tastypie.authentication.Authentication()`.

authorization

Controls which authorization class the `Resource` should use. Default is `tastypie.authorization.ReadOnlyAuthorization()`.

validation

Controls which validation class the `Resource` should use. Default is `tastypie.validation.Validation()`.

paginator_class

Controls which paginator class the `Resource` should use. Default is `tastypie.paginator.Paginator`.

Note: This is different than the other options in that you supply a class rather than an instance. This is done because the Paginator has some per-request initialization options.

---

cache

Controls which cache class the `Resource` should use. Default is `tastypie.cache.NoCache()`.

throttle

Controls which throttle class the `Resource` should use. Default is `tastypie.throttle.BaseThrottle()`.

allowed_methods

Controls what list & detail REST methods the `Resource` should respond to. Default is `None`, which means delegate to the more specific `list_allowed_methods` & `detail_allowed_methods` options.

You may specify a list like `['get', 'post', 'put', 'delete', 'patch']` as a shortcut to prevent having to specify the other options.

list_allowed_methods

Controls what list REST methods the `Resource` should respond to. Default is `['get', 'post', 'put', 'delete', 'patch']`. Set it to an empty list (i.e. *[]*) to disable all methods.

detail_allowed_methods

Controls what detail REST methods the `Resource` should respond to. Default is `['get', 'post', 'put', 'delete', 'patch']`. Set it to an empty list (i.e. *[]*) to disable all methods.

limit

Controls how many results the `Resource` will show at a time. Default is either the `API_LIMIT_PER_PAGE` setting (if provided) or `20` if not specified.

max_limit

Controls the maximum number of results the `Resource` will show at a time. If the user-specified `limit` is higher than this, it will be capped to this limit. Set to `0` or `None` to allow unlimited results.

api_name

An override for the `Resource` to use when generating resource URLs. Default is `None`.

resource_name

An override for the `Resource` to use when generating resource URLs. Default is `None`.

If not provided, the `Resource` or `ModelResource` will attempt to name itself. This means a lowercase version of the classname preceding the word `Resource` if present (i.e. `SampleContentResource` would become `samplecontent`).

default_format

Specifies the default serialization format the `Resource` should use if one is not requested (usually by the `Accept` header or `format` GET parameter). Default is `application/json`.

filtering

Specifies the fields that the `Resource` will accept client filtering on. Default is `{}`.

Keys should be the fieldnames as strings while values should be a list of accepted filter types.

This also restricts what fields can be filtered on when manually calling `obj_get` and `obj_get_list`.

ordering

Specifies the what fields the `Resource` should allow ordering on. Default is `[]`.

Values should be the fieldnames as strings. When provided to the `Resource` by the `order_by` GET parameter, you can specify either the `fieldname` (ascending order) or `-fieldname` (descending order).

object_class

Provides the `Resource` with the object that serves as the data source. Default is `None`.

In the case of `ModelResource`, this is automatically populated by the `queryset` option and is the model class.

queryset

Provides the `Resource` with the set of Django models to respond with. Default is `None`.

Unused by `Resource` but present for consistency.

> **Warning:** If you place any callables in this, they'll only be evaluated once (when the `Meta` class is instantiated). This especially affects things that are date/time related. Please see the *Tastypie Cookbook* for a way around this.

fields

Controls what introspected fields the `Resource` should include. A whitelist of fields. Default is `None`.

The default value of `None` means that all Django fields will be introspected. In order to specify that no fields should be introspected, use `[]`

excludes

Controls what introspected fields the `Resource` should *NOT* include. A blacklist of fields. Default is `[]`.

include_resource_uri

Specifies if the `Resource` should include an extra field that displays the detail URL (within the api) for that resource. Default is `True`.

include_absolute_url

Specifies if the `Resource` should include an extra field that displays the `get_absolute_url` for that object (on the site proper). Default is `False`.

always_return_data

Specifies all HTTP methods (except `DELETE`) should return a serialized form of the data. Default is `False`.

If `False`, `HttpNoContent` (204) is returned on `PUT` with an empty body & a `Location` header of where to request the full resource.

If `True`, `HttpResponse` (200) is returned on `POST/PUT` with a body containing all the data in a serialized form.

collection_name

Specifies the collection of objects returned in the `GET` list will be named. Default is `objects`.

detail_uri_name

Specifies the name for the regex group that matches on detail views. Defaults to `pk`.

Basic Filtering

`ModelResource` provides a basic Django ORM filter interface. Simply list the resource fields which you'd like to filter on and the allowed expression in a *filtering* property of your resource's Meta class:

```
from tastypie.constants import ALL, ALL_WITH_RELATIONS

class MyResource(ModelResource):
    class Meta:
        filtering = {
            "slug": ('exact', 'startswith',),
            "title": ALL,
        }
```

Valid filtering values are: Django ORM filters (e.g. `startswith`, `exact`, `lte`, etc.) or the `ALL` or `ALL_WITH_RELATIONS` constants defined in `tastypie.constants`.

These filters will be extracted from URL query strings using the same double-underscore syntax as the Django ORM:

```
/api/v1/myresource/?slug=myslug
/api/v1/myresource/?slug__startswith=test
```

Advanced Filtering

If you need to filter things other than ORM resources or wish to apply additional constraints (e.g. text filtering using django-haystack rather than simple database queries) your `Resource` may define a custom `build_filters()` method which allows you to filter the queryset before processing a request:

```
from haystack.query import SearchQuerySet

class MyResource(Resource):
    def build_filters(self, filters=None):
        if filters is None:
            filters = {}

        orm_filters = super(MyResource, self).build_filters(filters)

        if "q" in filters:
            sqs = SearchQuerySet().auto_query(filters['q'])

            orm_filters["pk__in"] = [i.pk for i in sqs]

        return orm_filters
```

Using PUT/DELETE/PATCH In Unsupported Places

Some places, like in certain browsers or hosts, don't allow the `PUT/DELETE/PATCH` methods. In these environments, you can simulate those kinds of requests by providing an `X-HTTP-Method-Override` header. For example, to send a `PATCH` request over `POST`, you'd send a request like:

```
curl --dump-header - -H "Content-Type: application/json" -H "X-HTTP-Method-Override:␣
↪PATCH" -X POST --data '{"title": "I Visited Grandma Today"}' http://localhost:8000/
↪api/v1/entry/1/
```

`Resource` Methods

Handles the data, request dispatch and responding to requests.

Serialization/deserialization is handled "at the edges" (i.e. at the beginning/end of the request/response cycle) so that everything internally is Python data structures.

This class tries to be non-model specific, so it can be hooked up to other data sources, such as search results, files, other data, etc.

`wrap_view`

`Resource.`**`wrap_view`**(*self*, *view*)

Wraps methods so they can be called in a more functional way as well as handling exceptions better.

Note that if `BadRequest` or an exception with a `response` attr are seen, there is special handling to either present a message back to the user or return the response traveling with the exception.

`get_response_class_for_exception`

`Resource.`**`get_response_class_for_exception`**(*self*, *request*, *exception*)

Can be overridden to customize response classes used for uncaught exceptions. Should always return a subclass of``django.http.HttpResponse``.

`base_urls`

`Resource.`**`base_urls`**(*self*)

The standard URLs this `Resource` should respond to. These include the list, detail, schema & multiple endpoints by default.

Should return a list of individual URLconf lines.

`override_urls`

`Resource.`**`override_urls`**(*self*)

Deprecated. Will be removed by v1.0.0. Please use `Resource.prepend_urls` instead.

prepend_urls

`Resource.`**`prepend_urls`**(*self*)

A hook for adding your own URLs or matching before the default URLs. Useful for adding custom endpoints or overriding the built-in ones (from `base_urls`).

Should return a list of individual URLconf lines.

urls

`Resource.`**`urls`**(*self*)

Property

The endpoints this `Resource` responds to. A combination of `base_urls` & `override_urls`.

Mostly a standard URLconf, this is suitable for either automatic use when registered with an `Api` class or for including directly in a URLconf should you choose to.

determine_format

`Resource.`**`determine_format`**(*self*, *request*)

Used to determine the desired format.

Largely relies on `tastypie.utils.mime.determine_format` but here as a point of extension.

serialize

`Resource.`**`serialize`**(*self*, *request*, *data*, *format*, *options=None*)

Given a request, data and a desired format, produces a serialized version suitable for transfer over the wire.

Mostly a hook, this uses the `Serializer` from `Resource._meta`.

deserialize

`Resource.`**`deserialize`**(*self*, *request*, *data*, *format='application/json'*)

Given a request, data and a format, deserializes the given data.

It relies on the request properly sending a `CONTENT_TYPE` header, falling back to `application/json` if not provided.

Mostly a hook, this uses the `Serializer` from `Resource._meta`.

alter_list_data_to_serialize

`Resource.`**`alter_list_data_to_serialize`**(*self*, *request*, *data*)

A hook to alter list data just before it gets serialized & sent to the user.

Useful for restructuring/renaming aspects of the what's going to be sent.

Should accommodate for a list of objects, generally also including meta data.

alter_detail_data_to_serialize

Resource.**alter_detail_data_to_serialize**(*self*, *request*, *data*)

A hook to alter detail data just before it gets serialized & sent to the user.

Useful for restructuring/renaming aspects of the what's going to be sent.

Should accommodate for receiving a single bundle of data.

alter_deserialized_list_data

Resource.**alter_deserialized_list_data**(*self*, *request*, *data*)

A hook to alter list data just after it has been received from the user & gets deserialized.

Useful for altering the user data before any hydration is applied.

alter_deserialized_detail_data

Resource.**alter_deserialized_detail_data**(*self*, *request*, *data*)

A hook to alter detail data just after it has been received from the user & gets deserialized.

Useful for altering the user data before any hydration is applied.

dispatch_list

Resource.**dispatch_list**(*self*, *request*, ***kwargs*)

A view for handling the various HTTP methods (GET/POST/PUT/DELETE) over the entire list of resources.

Relies on `Resource.dispatch` for the heavy-lifting.

dispatch_detail

Resource.**dispatch_detail**(*self*, *request*, ***kwargs*)

A view for handling the various HTTP methods (GET/POST/PUT/DELETE) on a single resource.

Relies on `Resource.dispatch` for the heavy-lifting.

dispatch

Resource.**dispatch**(*self*, *request_type*, *request*, ***kwargs*)

Handles the common operations (allowed HTTP method, authentication, throttling, method lookup) surrounding most CRUD interactions.

remove_api_resource_names

Resource.**remove_api_resource_names**(*self*, *url_dict*)

Given a dictionary of regex matches from a URLconf, removes `api_name` and/or `resource_name` if found.

This is useful for converting URLconf matches into something suitable for data lookup. For example:

```
Model.objects.filter(**self.remove_api_resource_names(matches))
```

method_check

Resource.**method_check**(*self*, *request*, *allowed=None*)

Ensures that the HTTP method used on the request is allowed to be handled by the resource.

Takes an `allowed` parameter, which should be a list of lowercase HTTP methods to check against. Usually, this looks like:

```
# The most generic lookup.
self.method_check(request, self._meta.allowed_methods)

# A lookup against what's allowed for list-type methods.
self.method_check(request, self._meta.list_allowed_methods)

# A useful check when creating a new endpoint that only handles
# GET.
self.method_check(request, ['get'])
```

is_authenticated

Resource.**is_authenticated**(*self*, *request*)

Handles checking if the user is authenticated and dealing with unauthenticated users.

Mostly a hook, this uses class assigned to `authentication` from `Resource._meta`.

throttle_check

Resource.**throttle_check**(*self*, *request*)

Handles checking if the user should be throttled.

Mostly a hook, this uses class assigned to `throttle` from `Resource._meta`.

log_throttled_access

Resource.**log_throttled_access**(*self*, *request*)

Handles the recording of the user's access for throttling purposes.

Mostly a hook, this uses class assigned to `throttle` from `Resource._meta`.

build_bundle

Resource.**build_bundle**(*self*, *obj=None*, *data=None*, *request=None*)

Given either an object, a data dictionary or both, builds a Bundle for use throughout the dehydrate/hydrate cycle.

If no object is provided, an empty object from Resource._meta.object_class is created so that attempts to access bundle.obj do not fail.

build_filters

Resource.**build_filters**(*self*, *filters=None*)

Allows for the filtering of applicable objects.

This needs to be implemented at the user level.

ModelResource includes a full working version specific to Django's Models.

apply_sorting

Resource.**apply_sorting**(*self*, *obj_list*, *options=None*)

Allows for the sorting of objects being returned.

This needs to be implemented at the user level.

ModelResource includes a full working version specific to Django's Models.

get_bundle_detail_data

Resource.**get_bundle_detail_data**(*self*, *bundle*)

Convenience method to return the detail_uri_name attribute off bundle.obj.

Usually just accesses bundle.obj.pk by default.

get_resource_uri

Resource.**get_resource_uri**(*self*, *bundle_or_obj=None*, *url_name='api_dispatch_list'*)

Handles generating a resource URI.

If the bundle_or_obj argument is not provided, it builds the URI for the list endpoint.

If the bundle_or_obj argument is provided, it builds the URI for the detail endpoint.

Return the generated URI. If that URI can not be reversed (not found in the URLconf), it will return an empty string.

resource_uri_kwargs

Resource.**resource_uri_kwargs**(*self*, *bundle_or_obj=None*)

Handles generating a resource URI.

If the `bundle_or_obj` argument is not provided, it builds the URI for the list endpoint.

If the `bundle_or_obj` argument is provided, it builds the URI for the detail endpoint.

Return the generated URI. If that URI can not be reversed (not found in the URLconf), it will return `None`.

detail_uri_kwargs

`Resource.`**`detail_uri_kwargs`**(*self*, *bundle_or_obj*)

This needs to be implemented at the user level.

Given a `Bundle` or an object, it returns the extra kwargs needed to generate a detail URI.

`ModelResource` includes a full working version specific to Django's `Models`.

get_via_uri

`Resource.`**`get_via_uri`**(*self*, *uri*, *request=None*)

This pulls apart the salient bits of the URI and populates the resource via a `obj_get`.

Optionally accepts a `request`.

If you need custom behavior based on other portions of the URI, simply override this method.

full_dehydrate

`Resource.`**`full_dehydrate`**(*self*, *bundle*, *for_list=False*)

Populate the bundle's `data` attribute.

The `bundle` parameter will have the data that needs dehydrating in its `obj` attribute.

The `for_list` parameter indicates the style of response being prepared:

- `True` indicates a list of items. Note that `full_dehydrate()` will be called once for each object requested.
- `False` indicates a response showing the details for an item

This method is responsible for invoking the the `dehydrate()` method of all the fields in the resource. Additionally, it calls *`Resource.dehydrate()`*.

Must return a `Bundle` with the desired dehydrated `data` (usually a `dict`). Typically one should modify the bundle passed in and return it, but you may also return a completely new bundle.

dehydrate

`Resource.`**`dehydrate`**(*self*, *bundle*)

A hook to allow a final manipulation of data once all fields/methods have built out the dehydrated data.

Useful if you need to access more than one dehydrated field or want to annotate on additional data.

Must return the modified bundle.

full_hydrate

`Resource.`**`full_hydrate`**(*self*, *bundle*)

Given a populated bundle, distill it and turn it back into a full-fledged object instance.

hydrate

`Resource.`**`hydrate`**(*self*, *bundle*)

A hook to allow a final manipulation of data once all fields/methods have built out the hydrated data.

Useful if you need to access more than one hydrated field or want to annotate on additional data.

Must return the modified bundle.

hydrate_m2m

`Resource.`**`hydrate_m2m`**(*self*, *bundle*)

Populate the ManyToMany data on the instance.

build_schema

`Resource.`**`build_schema`**(*self*)

Returns a dictionary of all the fields on the resource and some properties about those fields.

Used by the `schema/` endpoint to describe what will be available.

dehydrate_resource_uri

`Resource.`**`dehydrate_resource_uri`**(*self*, *bundle*)

For the automatically included `resource_uri` field, dehydrate the URI for the given bundle.

Returns empty string if no URI can be generated.

generate_cache_key

`Resource.`**`generate_cache_key`**(*self*, **args*, ***kwargs*)

Creates a unique-enough cache key.

This is based off the current api_name/resource_name/args/kwargs.

get_object_list

`Resource.`**`get_object_list`**(*self*, *request*)

A hook to allow making returning the list of available objects.

This needs to be implemented at the user level.

`ModelResource` includes a full working version specific to Django's `Models`.

can_create

Resource.**can_create**(*self*)

Checks to ensure `post` is within `allowed_methods`.

can_update

Resource.**can_update**(*self*)

Checks to ensure `put` is within `allowed_methods`.

Used when hydrating related data.

can_delete

Resource.**can_delete**(*self*)

Checks to ensure `delete` is within `allowed_methods`.

apply_filters

Resource.**apply_filters**(*self*, *request*, *applicable_filters*)

A hook to alter how the filters are applied to the object list.

This needs to be implemented at the user level.

`ModelResource` includes a full working version specific to Django's `Models`.

obj_get_list

Resource.**obj_get_list**(*self*, *bundle*, ***kwargs*)

Fetches the list of objects available on the resource.

This needs to be implemented at the user level.

`ModelResource` includes a full working version specific to Django's `Models`.

cached_obj_get_list

Resource.**cached_obj_get_list**(*self*, *bundle*, ***kwargs*)

A version of `obj_get_list` that uses the cache as a means to get commonly-accessed data faster.

obj_get

Resource.**obj_get**(*self*, *bundle*, ***kwargs*)

Fetches an individual object on the resource.

This needs to be implemented at the user level. If the object can not be found, this should raise a `NotFound` exception.

`ModelResource` includes a full working version specific to Django's `Models`.

cached_obj_get

`Resource.`**`cached_obj_get`**(*self*, *bundle*, ***kwargs*)

A version of `obj_get` that uses the cache as a means to get commonly-accessed data faster.

obj_create

`Resource.`**`obj_create`**(*self*, *bundle*, ***kwargs*)

Creates a new object based on the provided data.

This needs to be implemented at the user level.

`ModelResource` includes a full working version specific to Django's `Models`.

lookup_kwargs_with_identifiers

`Resource.`**`lookup_kwargs_with_identifiers`**(*self*, *bundle*, *kwargs*)

Kwargs here represent uri identifiers. Ex: /repos/<user_id>/<repo_name>/ We need to turn those identifiers into Python objects for generating lookup parameters that can find them in the DB.

obj_update

`Resource.`**`obj_update`**(*self*, *bundle*, ***kwargs*)

Updates an existing object (or creates a new object) based on the provided data.

This needs to be implemented at the user level.

`ModelResource` includes a full working version specific to Django's `Models`.

obj_delete_list

`Resource.`**`obj_delete_list`**(*self*, *bundle*, ***kwargs*)

Deletes an entire list of objects.

This needs to be implemented at the user level.

`ModelResource` includes a full working version specific to Django's `Models`.

obj_delete_list_for_update

`Resource.`**`obj_delete_list_for_update`**(*self*, *bundle*, ***kwargs*)

Deletes an entire list of objects, specific to PUT list.

This needs to be implemented at the user level.

`ModelResource` includes a full working version specific to Django's `Models`.

obj_delete

Resource.**obj_delete**(*self*, *bundle*, ***kwargs*)

Deletes a single object.

This needs to be implemented at the user level.

ModelResource includes a full working version specific to Django's Models.

create_response

Resource.**create_response**(*self*, *request*, *data*, *response_class=HttpResponse*, ***response_kwargs*)

Extracts the common "which-format/serialize/return-response" cycle.

Mostly a useful shortcut/hook.

is_valid

Resource.**is_valid**(*self*, *bundle*)

Handles checking if the data provided by the user is valid.

Mostly a hook, this uses class assigned to validation from Resource._meta.

If validation fails, an error is raised with the error messages serialized inside it.

rollback

Resource.**rollback**(*self*, *bundles*)

Given the list of bundles, delete all objects pertaining to those bundles.

This needs to be implemented at the user level. No exceptions should be raised if possible.

ModelResource includes a full working version specific to Django's Models.

get_list

Resource.**get_list**(*self*, *request*, ***kwargs*)

Returns a serialized list of resources.

Calls obj_get_list to provide the data, then handles that result set and serializes it.

Should return a HttpResponse (200 OK).

get_detail

Resource.**get_detail**(*self*, *request*, ***kwargs*)

Returns a single serialized resource.

Calls cached_obj_get/obj_get to provide the data, then handles that result set and serializes it.

Should return a HttpResponse (200 OK).

put_list

Resource.**put_list**(*self*, *request*, ***kwargs*)

Replaces a collection of resources with another collection.

Calls `delete_list` to clear out the collection then `obj_create` with the provided the data to create the new collection.

Return `HttpNoContent` (204 No Content) if `Meta.always_return_data = False` (default).

Return `HttpAccepted` (200 OK) if `Meta.always_return_data = True`.

put_detail

Resource.**put_detail**(*self*, *request*, ***kwargs*)

Either updates an existing resource or creates a new one with the provided data.

Calls `obj_update` with the provided data first, but falls back to `obj_create` if the object does not already exist.

If a new resource is created, return `HttpCreated` (201 Created). If `Meta.always_return_data = True`, there will be a populated body of serialized data.

If an existing resource is modified and `Meta.always_return_data = False` (default), return `HttpNoContent` (204 No Content). If an existing resource is modified and `Meta.always_return_data = True`, return `HttpAccepted` (200 OK).

post_list

Resource.**post_list**(*self*, *request*, ***kwargs*)

Creates a new resource/object with the provided data.

Calls `obj_create` with the provided data and returns a response with the new resource's location.

If a new resource is created, return `HttpCreated` (201 Created). If `Meta.always_return_data = True`, there will be a populated body of serialized data.

post_detail

Resource.**post_detail**(*self*, *request*, ***kwargs*)

Creates a new subcollection of the resource under a resource.

This is not implemented by default because most people's data models aren't self-referential.

If a new resource is created, return `HttpCreated` (201 Created).

delete_list

Resource.**delete_list**(*self*, *request*, ***kwargs*)

Destroys a collection of resources/objects.

Calls `obj_delete_list`.

If the resources are deleted, return `HttpNoContent` (204 No Content).

delete_detail

Resource.**delete_detail**(*self*, *request*, ***kwargs*)

Destroys a single resource/object.

Calls obj_delete.

If the resource is deleted, return HttpNoContent (204 No Content). If the resource did not exist, return HttpNotFound (404 Not Found).

patch_list

Resource.**patch_list**(*self*, *request*, ***kwargs*)

Updates a collection in-place.

The exact behavior of PATCH to a list resource is still the matter of some debate in REST circles, and the PATCH RFC isn't standard. So the behavior this method implements (described below) is something of a stab in the dark. It's mostly cribbed from GData, with a smattering of ActiveResource-isms and maybe even an original idea or two.

The PATCH format is one that's similar to the response returned from a GET on a list resource:

```
{
  "objects": [{object}, {object}, ...],
  "deleted_objects": ["URI", "URI", "URI", ...],
}
```

For each object in objects:

- If the dict does not have a resource_uri key then the item is considered "new" and is handled like a POST to the resource list.
- If the dict has a resource_uri key and the resource_uri refers to an existing resource then the item is an update; it's treated like a PATCH to the corresponding resource detail.
- If the dict has a resource_uri but the resource *doesn't* exist, then this is considered to be a create-via-PUT.

Each entry in deleted_objects refers to a resource URI of an existing resource to be deleted; each is handled like a DELETE to the relevant resource.

In any case:

- If there's a resource URI it *must* refer to a resource of this type. It's an error to include a URI of a different resource.
- PATCH is all or nothing. If a single sub-operation fails, the entire request will fail and all resources will be rolled back.
- For PATCH to work, you **must** have patch in your *detail_allowed_methods* setting.
- To delete objects via deleted_objects in a PATCH request you **must** have delete in your *detail_allowed_methods* setting.

patch_detail

Resource.**patch_detail**(*self*, *request*, ***kwargs*)

Updates a resource in-place.

Calls obj_update.

If the resource is updated, return `HttpAccepted` (202 Accepted). If the resource did not exist, return `HttpNotFound` (404 Not Found).

get_schema

`Resource.`**`get_schema`**(*self*, *request*, ***kwargs*)

Returns a serialized form of the schema of the resource.

Calls `build_schema` to generate the data. This method only responds to HTTP GET.

Should return a HttpResponse (200 OK).

get_multiple

`Resource.`**`get_multiple`**(*self*, *request*, ***kwargs*)

Returns a serialized list of resources based on the identifiers from the URL.

Calls `obj_get` to fetch only the objects requested. This method only responds to HTTP GET.

Should return a HttpResponse (200 OK).

ModelResource Methods

A subclass of `Resource` designed to work with Django's `Models`.

This class will introspect a given `Model` and build a field list based on the fields found on the model (excluding relational fields).

Given that it is aware of Django's ORM, it also handles the CRUD data operations of the resource.

should_skip_field

`ModelResource.`**`should_skip_field`**(*cls*, *field*)

Class method

Given a Django model field, return if it should be included in the contributed ApiFields.

api_field_from_django_field

`ModelResource.`**`api_field_from_django_field`**(*cls*, *f*, *default=CharField*)

Class method

Returns the field type that would likely be associated with each Django type.

get_fields

`ModelResource.`**`get_fields`**(*cls*, *fields=None*, *excludes=None*)

Class method

Given any explicit fields to include and fields to exclude, add additional fields based on the associated model.

check_filtering

ModelResource.**check_filtering**(*self*, *field_name*, *filter_type='exact'*, *filter_bits=None*)

Given a field name, an optional filter type and an optional list of additional relations, determine if a field can be filtered on.

If a filter does not meet the needed conditions, it should raise an `InvalidFilterError`.

If the filter meets the conditions, a list of attribute names (not field names) will be returned.

build_filters

ModelResource.**build_filters**(*self*, *filters=None*)

Given a dictionary of filters, create the necessary ORM-level filters.

Keys should be resource fields, **NOT** model fields.

Valid values are either a list of Django filter types (i.e. `['startswith', 'exact', 'lte']`), the `ALL` constant or the `ALL_WITH_RELATIONS` constant.

At the declarative level:

```
filtering = {
    'resource_field_name': ['exact', 'startswith', 'endswith', 'contains'],
    'resource_field_name_2': ['exact', 'gt', 'gte', 'lt', 'lte', 'range'],
    'resource_field_name_3': ALL,
    'resource_field_name_4': ALL_WITH_RELATIONS,
    ...
}
```

Accepts the filters as a dict. `None` by default, meaning no filters.

apply_sorting

ModelResource.**apply_sorting**(*self*, *obj_list*, *options=None*)

Given a dictionary of options, apply some ORM-level sorting to the provided `QuerySet`.

Looks for the `order_by` key and handles either ascending (just the field name) or descending (the field name with a `-` in front).

The field name should be the resource field, **NOT** model field.

apply_filters

ModelResource.**apply_filters**(*self*, *request*, *applicable_filters*)

An ORM-specific implementation of `apply_filters`.

The default simply applies the `applicable_filters` as `**kwargs`, but should make it possible to do more advanced things.

get_object_list

ModelResource.**get_object_list**(*self*, *request*)

A ORM-specific implementation of `get_object_list`.

Returns a `QuerySet` that may have been limited by other overrides.

obj_get_list

ModelResource.**obj_get_list**(*self*, *filters=None*, ***kwargs*)

A ORM-specific implementation of `obj_get_list`.

Takes an optional `filters` dictionary, which can be used to narrow the query.

obj_get

ModelResource.**obj_get**(*self*, ***kwargs*)

A ORM-specific implementation of `obj_get`.

Takes optional `kwargs`, which are used to narrow the query to find the instance.

obj_create

ModelResource.**obj_create**(*self*, *bundle*, ***kwargs*)

A ORM-specific implementation of `obj_create`.

obj_update

ModelResource.**obj_update**(*self*, *bundle*, ***kwargs*)

A ORM-specific implementation of `obj_update`.

obj_delete_list

ModelResource.**obj_delete_list**(*self*, ***kwargs*)

A ORM-specific implementation of `obj_delete_list`.

Takes optional `kwargs`, which can be used to narrow the query.

obj_delete_list_for_update

ModelResource.**obj_delete_list_for_update**(*self*, ***kwargs*)

A ORM-specific implementation of `obj_delete_list_for_update`.

Takes optional `kwargs`, which can be used to narrow the query.

obj_delete

ModelResource.**obj_delete**(*self*, ***kwargs*)

A ORM-specific implementation of `obj_delete`.

Takes optional `kwargs`, which are used to narrow the query to find the instance.

rollback

ModelResource.**rollback**(*self*, *bundles*)

A ORM-specific implementation of `rollback`.

Given the list of bundles, delete all models pertaining to those bundles.

save_related

ModelResource.**save_related**(*self*, *bundle*)

Handles the saving of related non-M2M data.

Calling assigning `child.parent = parent` & then calling `Child.save` isn't good enough to make sure the `parent` is saved.

To get around this, we go through all our related fields & call `save` on them if they have related, non-M2M data. M2M data is handled by the `ModelResource.save_m2m` method.

save_m2m

ModelResource.**save_m2m**(*self*, *bundle*)

Handles the saving of related M2M data.

Due to the way Django works, the M2M data must be handled after the main instance, which is why this isn't a part of the main `save` bits.

Currently slightly inefficient in that it will clear out the whole relation and recreate the related data as needed.

get_resource_uri

ModelResource.**get_resource_uri**(*self*, *bundle_or_obj*)

Handles generating a resource URI for a single resource.

Uses the model's `pk` in order to create the URI.

CHAPTER 10

Bundles

What Are Bundles?

Bundles are a small abstraction that allow Tastypie to pass data between resources. This allows us not to depend on passing `request` to every single method (especially in places where this would be overkill). It also allows resources to work with data coming into the application paired together with an unsaved instance of the object in question. Finally, it aids in keeping Tastypie more thread-safe.

Think of it as package of user data & an object instance (either of which are optionally present).

Attributes

All data within a bundle can be optional, especially depending on how it's being used. If you write custom code using `Bundle`, make sure appropriate guards are in place.

`obj`

Either a Python object or `None`.

Usually a Django model, though it may/may not have been saved already.

`data`

Always a plain Python dictionary of data. If not provided, it will be empty.

`request`

Either the Django `request` that's part of the issued request or an empty `HttpRequest` if it wasn't provided.

related_obj

Either another "parent" Python object or `None`.

Useful when handling one-to-many relations. Used in conjunction with `related_name`.

related_name

Either a Python string name of an attribute or `None`.

Useful when handling one-to-many relations. Used in conjunction with `related_obj`.

CHAPTER 11

Api

In terms of a REST-style architecture, the "api" is a collection of resources. In Tastypie, the `Api` gathers together the `Resources` & provides a nice way to use them as a set. It handles many of the URLconf details for you, provides a helpful "top-level" view to show what endpoints are available & some extra URL resolution juice.

Quick Start

A sample api definition might look something like (usually located in a URLconf):

```
from django.conf.urls import url, include
from tastypie.api import Api
from myapp.api.resources import UserResource, EntryResource

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

# Standard bits...
urlpatterns = [
    url(r'^api/', include(v1_api.urls)),
]
```

For namespaced urls see *Namespaces*

`Api` Methods

Implements a registry to tie together the various resources that make up an API.

Especially useful for navigation, HATEOAS and for providing multiple versions of your API.

Optionally supplying `api_name` allows you to name the API. Generally, this is done with version numbers (i.e. `v1`, `v2`, etc.) but can be named any string.

register

Api.register(self, resource, canonical=True):

Registers an instance of a `Resource` subclass with the API.

Optionally accept a `canonical` argument, which indicates that the resource being registered is the canonical variant. Defaults to `True`.

unregister

Api.unregister(self, resource_name):

If present, unregisters a resource from the API.

canonical_resource_for

Api.canonical_resource_for(self, resource_name):

Returns the canonical resource for a given `resource_name`.

override_urls

Api.override_urls(self):

Deprecated. Will be removed by v1.0.0. Please use `Api.prepend_urls` instead.

prepend_urls

Api.prepend_urls(self):

A hook for adding your own URLs or matching before the default URLs. Useful for adding custom endpoints or overriding the built-in ones.

Should return a list of individual URLconf lines.

urls

Api.urls(self):

Property

Provides URLconf details for the `Api` and all registered `Resources` beneath it.

top_level

Api.top_level(self, request, api_name=None):

A view that returns a serialized list of all resources registers to the `Api`. Useful for discovery.

CHAPTER 12

Resource Fields

When designing an API, an important component is defining the representation of the data you're presenting. Like Django models, you can control the representation of a `Resource` using fields. There are a variety of fields for various types of data.

Quick Start

For the impatient:

```
from tastypie import fields, utils
from tastypie.resources import Resource
from myapp.api.resources import ProfileResource, NoteResource


class PersonResource(Resource):
    name = fields.CharField(attribute='name')
    age = fields.IntegerField(attribute='years_old', null=True)
    created = fields.DateTimeField(readonly=True, help_text='When the person was␣
↪created', default=utils.now)
    is_active = fields.BooleanField(default=True)
    profile = fields.ToOneField(ProfileResource, 'profile')
    notes = fields.ToManyField(NoteResource, 'notes', full=True)
```

Standard Data Fields

All standard data fields have a common base class `ApiField` which handles the basic implementation details.

Note: You should not use the `ApiField` class directly. Please use one of the subclasses that is more correct for your data.

Common Field Options

All `ApiField` objects accept the following options.

attribute

`ApiField.`**`attribute`**

A string naming an instance attribute of the object wrapped by the Resource. The attribute will be accessed during the `dehydrate` or or written during the `hydrate`.

Defaults to `None`, meaning data will be manually accessed.

default

`ApiField.`**`default`**

Provides default data when the object being `dehydrated`/`hydrated` has no data on the field.

Defaults to `tastypie.fields.NOT_PROVIDED`.

null

`ApiField.`**`null`**

Indicates whether or not a `None` is allowable data on the field. Defaults to `False`.

blank

`ApiField.`**`blank`**

Indicates whether or not data may be omitted on the field. Defaults to `False`.

This is useful for allowing the user to omit data that you can populate based on the request, such as the `user` or `site` to associate a record with.

readonly

`ApiField.`**`readonly`**

Indicates whether the field is used during the `hydrate` or not. Defaults to `False`.

unique

`ApiField.`**`unique`**

Indicates whether the field is a unique identifier for the object.

help_text

`ApiField.`**`help_text`**

A human-readable description of the field exposed at the schema level. Defaults to the per-Field definition.

use_in

ApiField.**use_in**

Optionally omit this field in list or detail views. It can be either 'all', 'list', or 'detail' or a callable which accepts a bundle and returns a boolean value.

Field Types

BooleanField

A boolean field.

Covers both `models.BooleanField` and `models.NullBooleanField`.

CharField

A text field of arbitrary length.

Covers both `models.CharField` and `models.TextField`.

DateField

A date field.

DateTimeField

A datetime field.

DecimalField

A decimal field.

DictField

A dictionary field.

FileField

A file-related field.

Covers both `models.FileField` and `models.ImageField`.

FloatField

A floating point field.

IntegerField

An integer field.

Covers `models.IntegerField`, `models.PositiveIntegerField`, `models.PositiveSmallIntegerField` and `models.SmallIntegerField`.

ListField

A list field.

TimeField

A time field.

Relationship Fields

Provides access to data that is related within the database.

The `RelatedField` base class is not intended for direct use but provides functionality that `ToOneField` and `ToManyField` build upon.

The contents of this field actually point to another `Resource`, rather than the related object. This allows the field to represent its data in different ways.

The abstractions based around this are "leaky" in that, unlike the other fields provided by `tastypie`, these fields don't handle arbitrary objects very well. The subclasses use Django's ORM layer to make things go, though there is no ORM-specific code at this level.

Common Field Options

In addition to the common attributes for all *ApiField*, relationship fields accept the following.

to

`RelatedField.`**`to`**

The `to` argument should point to a `Resource` class, NOT to a `Model`. Required.

full

`RelatedField.`**`full`**

Indicates how the related `Resource` will appear post-`dehydrate`. If `False`, the related `Resource` will appear as a URL to the endpoint of that resource. If `True`, the result of the sub-resource's `dehydrate` will be included in full. You can further control post-`dehydrate` behaviour when requesting a resource or a list of resources by setting `full_list` and `full_detail`.

full_list

RelatedField.**full_list**

Indicates how the related Resource will appear post-dehydrate when requesting a list of resources. The value is one of True, False or a callable that accepts a bundle and returns True or False. If False, the related Resource will appear as a URL to the endpoint of that resource if accessing a list of resources. If True and full is also True, the result of thesub-resource's dehydrate will be included in full. Default is True

full_detail

RelatedField.**full_detail**

Indicates how the related Resource will appear post-dehydrate when requesting a single resource. The value is one of True, False or a callable that accepts a bundle and returns True or False. If False, the related Resource will appear as a URL to the endpoint of that resource if accessing a specific resources. If True and full is also True, the result of thesub-resource's dehydrate will be included in full. Default is True

related_name

RelatedField.**related_name**

Used to help automatically populate reverse relations when creating data. Defaults to None.

In order for this option to work correctly, there must be a field on the other Resource with this as an attribute/instance_name. Usually this just means adding a reflecting ToOneField pointing back.

Example:

```
class EntryResource(ModelResource):
    authors = fields.ToManyField('path.to.api.resources.AuthorResource', 'author_set',
↪ related_name='entry')

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

class AuthorResource(ModelResource):
    entry = fields.ToOneField(EntryResource, 'entry')

    class Meta:
        queryset = Author.objects.all()
        resource_name = 'author'
```

Field Types

ToOneField

Provides access to related data via foreign key.

This subclass requires Django's ORM layer to work properly.

OneToOneField

An alias to ToOneField for those who prefer to mirror django.db.models.

ForeignKey

An alias to `ToOneField` for those who prefer to mirror `django.db.models`.

ToManyField

Provides access to related data via a join table.

This subclass requires Django's ORM layer to work properly.

This field also has special behavior when dealing with `attribute` in that it can take a callable. For instance, if you need to filter the reverse relation, you can do something like:

```
subjects = fields.ToManyField(SubjectResource, attribute=lambda bundle: Subject.
↪objects.filter(notes=bundle.obj, name__startswith='Personal'))
```

The callable should either return an iterable of objects or `None`.

Note that the `hydrate` portions of this field are quite different than any other field. `hydrate_m2m` actually handles the data and relations. This is due to the way Django implements M2M relationships.

ManyToManyField

An alias to `ToManyField` for those who prefer to mirror `django.db.models`.

OneToManyField

An alias to `ToManyField` for those who prefer to mirror `django.db.models`.

CHAPTER 13

Caching

When adding an API to your site, it's important to understand that most consumers of the API will not be people, but instead machines. This means that the traditional "fetch-read-click" cycle is no longer measured in minutes but in seconds or milliseconds.

As such, caching is a very important part of the deployment of your API. Tastypie ships with two classes to make working with caching easier. These caches store at the object level, reducing access time on the database.

However, it's worth noting that these do *NOT* cache serialized representations. For heavy traffic, we'd encourage the use of a caching proxy, especially Varnish, as it shines under this kind of usage. It's far faster than Django views and already neatly handles most situations.

The first section below demonstrates how to cache at the Django level, reducing the amount of work required to satisfy a request. In many cases your API serves web browsers or is behind by a caching proxy such as Varnish and it is possible to set HTTP Cache-Control headers to avoid issuing a request to your application at all. This is discussed in the *HTTP Cache-Control* section below.

Usage

Using these classes is simple. Simply provide them (or your own class) as a `Meta` option to the `Resource` in question. For example:

```
from django.contrib.auth.models import User
from tastypie.cache import SimpleCache
from tastypie.resources import ModelResource


class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
        resource_name = 'auth/user'
        excludes = ['email', 'password', 'is_superuser']
        # Add it here.
        cache = SimpleCache(timeout=10)
```

Caching Options

Tastypie ships with the following `Cache` classes:

NoCache

The no-op cache option, this does no caching but serves as an api-compatible plug. Very useful for development.

SimpleCache

This option does basic object caching, attempting to find the object in the cache & writing the object to the cache. By default, it uses the `default` cache backend as configured in the `CACHES` setting. However, an optional *cache_name* parameter can be passed to the constructor to specify a different backend. For example, if `CACHES` looks like:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
        'TIMEOUT': 60
    },
    'resources': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
        'TIMEOUT': 60
    }
}
```

you can configure your resource's *cache_name* property like so:

```
cache = SimpleCache(cache_name='resources', timeout=10)
```

In this case, the cache used will be the one named, and the default *timeout* specified in `CACHES['resources']` will be overriden by the *timeout* parameter.

Implementing Your Own Cache

Implementing your own `Cache` class is as simple as subclassing `NoCache` and overriding the `get` & `set` methods. For example, a json-backed cache might look like:

```
import json
from django.conf import settings
from tastypie.cache import NoCache


class JSONCache(NoCache):
    def _load(self):
        data_file = open(settings.TASTYPIE_JSON_CACHE, 'r')
        return json.load(data_file)

    def _save(self, data):
        data_file = open(settings.TASTYPIE_JSON_CACHE, 'w')
        return json.dump(data, data_file)

    def get(self, key):
```

```
        data = self._load()
        return data.get(key, None)

    def set(self, key, value, timeout=60):
        data = self._load()
        data[key] = value
        self._save(data)
```

Note that this is *NOT* necessarily an optimal solution, but is simply demonstrating how one might go about implementing your own `Cache`.

HTTP Cache-Control

The HTTP protocol defines a `Cache-Control` header, which can be used to tell clients and intermediaries who is allowed to cache a response and for how long. Mark Nottingham has a general caching introduction and the Django cache documentation describes how to set caching-related headers in your code. The range of possible options is beyond the scope of this documentation, but it's important to know that, by default, Tastypie will prevent responses from being cached to ensure that clients always receive current information.

To override the default `no-cache` response, your `Resource` should ensure that your `cache` class implements `cache_control`. The default `SimpleCache` does this by default. It uses the timeout passed to the initialization as the `max-age` and `s-maxage`. By default, it does not claim to know if the results should be public or privately cached but this can be changed by passing either a `public=True` or a `private=True` to the initialization of the `SimpleClass`.

Behind the scenes, the return value from the `cache_control` method is passed to the cache_control helper provided by Django. If you wish to add your own methods to it, you can do so by overloading the `cache_control` method and modifying the dictionary it returns.:

```
from tastypie.cache import SimpleCache

class NoTransformCache(SimpleCache):

    def cache_control(self):
        control = super(NoTransformCache, self).cache_control()
        control.update({"no_transform": True})
        return control
```

HTTP Vary

The HTTP protocol defines a `Vary` header, which can be used to tell clients and intermediaries on what headers your response varies. This allows clients to store a correct response for each type. By default, Tastypie will send the `Vary: Accept` header so that a seperate response is cached for each `Content-Type`. However, if you wish to change this, simply pass a list to the `varies` kwarg of any `Cache` class.

It is important to note that if a list is passed, Tastypie not automatically include the `Vary: Accept` and you should include it as a member of your list.:

```
class ExampleResource(Resource):
    class Meta:
        cache = SimpleCache(varies=["Accept", "Cookie"])
```

CHAPTER 14

Validation

Validation allows you to ensure that the data being submitted by the user is appropriate for storage. This can range from simple type checking on up to complex validation that compares different fields together.

If the data is valid, an empty dictionary is returned and processing continues as normal. If the data is invalid, a dictionary of error messages (keys being the field names, values being a list of error messages) is immediately returned to the user, serialized in the format they requested.

Usage

Using these classes is simple. Simply provide them (or your own class) as a `Meta` option to the `Resource` in question. For example:

```
from django.contrib.auth.models import User
from tastypie.validation import Validation
from tastypie.resources import ModelResource


class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
        resource_name = 'auth/user'
        excludes = ['email', 'password', 'is_superuser']
        # Add it here.
        validation = Validation()
```

Validation Options

Tastypie ships with the following `Validation` classes:

Validation

The no-op validation option, the data submitted is always considered to be valid.

This is the default class hooked up to `Resource/ModelResource`.

FormValidation

A more complex form of validation, this class accepts a `form_class` argument to its constructor. You supply a Django `Form` (or `ModelForm`, though `save` will never get called) and Tastypie will verify the `data` in the `Bundle` against the form.

This class **DOES NOT** alter the data sent, only verifies it. If you want to alter the data, please use the `CleanedDataFormValidation` class instead.

> **Warning:** Data in the bundle must line up with the fieldnames in the `Form`. If they do not, you'll need to either munge the data or change your form.

Usage looks like:

```
from django import forms

class NoteForm(forms.Form):
    title = forms.CharField(max_length=100)
    slug = forms.CharField(max_length=50)
    content = forms.CharField(required=False, widget=forms.Textarea)
    is_active = forms.BooleanField()

form = FormValidation(form_class=NoteForm)
```

CleanedDataFormValidation

Similar to the `FormValidation` class, this uses a Django `Form` to handle validation. **However**, it will use the `form.cleaned_data` to replace the `bundle` data sent by user! Usage is identical to `FormValidation`.

Implementing Your Own Validation

Implementing your own `Validation` classes is a simple process. The constructor can take whatever `**kwargs` it needs (if any). The only other method to implement is the `is_valid` method:

```
from tastypie.validation import Validation


class AwesomeValidation(Validation):
    def is_valid(self, bundle, request=None):
        if not bundle.data:
            return {'__all__': 'Not quite what I had in mind.'}

        errors = {}

        for key, value in bundle.data.items():
            if not isinstance(value, basestring):
```

```
                continue

            if not 'awesome' in value:
                errors[key] = ['NOT ENOUGH AWESOME. NEEDS MORE.']

        return errors
```

Under this validation, every field that's a string is checked for the word 'awesome'. If it's not in the string, it's an error.

CHAPTER 15

Authentication

Authentication is the component needed to verify who a certain user is and to validate their access to the API.

Authentication answers the question "Who is this person?" This usually involves requiring credentials, such as an API key or username/password or oAuth tokens.

Usage

Using these classes is simple. Simply provide them (or your own class) as a `Meta` option to the `Resource` in question. For example:

```
from django.contrib.auth.models import User
from tastypie.authentication import BasicAuthentication
from tastypie.resources import ModelResource


class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
        resource_name = 'auth/user'
        excludes = ['email', 'password', 'is_superuser']
        # Add it here.
        authentication = BasicAuthentication()
```

Authentication Options

Tastypie ships with the following `Authentication` classes:

Authentication

The no-op authentication option, the client is always allowed through. Very useful for development and read-only APIs.

BasicAuthentication

This authentication scheme uses HTTP Basic Auth to check a user's credentials. The username is their `django.contrib.auth.models.User` username (assuming it is present) and their password should also correspond to that entry.

> **Warning:** If you're using Apache & `mod_wsgi`, you will need to enable `WSGIPassAuthorization On`. See this post for details.

ApiKeyAuthentication

As an alternative to requiring sensitive data like a password, the `ApiKeyAuthentication` allows you to collect just username & a machine-generated api key. Tastypie ships with a special `Model` just for this purpose, so you'll need to ensure `tastypie` is in `INSTALLED_APPS` and that the model's database tables have been created (e.g. via `django-admin.py syncdb`).

To use this mechanism, the end user can either specify an `Authorization` header or pass the `username/api_key` combination as `GET/POST` parameters. Examples:

```
# As a header
# Format is ``Authorization: ApiKey <username>:<api_key>
Authorization: ApiKey daniel:204db7bcfafb2deb7506b89eb3b9b715b09905c8

# As GET params
http://127.0.0.1:8000/api/v1/entries/?username=daniel&api_
↪key=204db7bcfafb2deb7506b89eb3b9b715b09905c8
```

Tastypie includes a signal function you can use to auto-create `ApiKey` objects. Hooking it up looks like:

```
from django.contrib.auth.models import User
from django.db.models import signals
from tastypie.models import create_api_key

signals.post_save.connect(create_api_key, sender=User)
```

> **Warning:** If you're using Apache & `mod_wsgi`, you will need to enable `WSGIPassAuthorization On`, otherwise `mod_wsgi` strips out the `Authorization` header. See this post for details (even though it only mentions Basic auth).

Note: In some cases it may be useful to make the `ApiKey` model an abstract base class. To enable this, set `settings.TASTYPIE_ABSTRACT_APIKEY` to `True`. See the documentation for this setting for more information.

SessionAuthentication

This authentication scheme uses the built-in Django sessions to check if a user is logged. This is typically useful when used by Javascript on the same site as the API is hosted on.

It requires that the user has logged in & has an active session. They also must have a valid CSRF token.

DigestAuthentication

This authentication scheme uses HTTP Digest Auth to check a user's credentials. The username is their `django.contrib.auth.models.User` username (assuming it is present) and their password should be their machine-generated api key. As with ApiKeyAuthentication, `tastypie` should be included in `INSTALLED_APPS`.

> **Warning:** If you're using Apache & `mod_wsgi`, you will need to enable `WSGIPassAuthorization On`. See this post for details (even though it only mentions Basic auth).

OAuthAuthentication

Handles OAuth, which checks a user's credentials against a separate service. Currently verifies against OAuth 1.0a services.

This does *NOT* provide OAuth authentication in your API, strictly consumption.

> **Warning:** If you're used to in-browser OAuth flow (click a "Sign In" button, get redirected, login on remote service, get redirected back), this isn't the same. Most prominently, expecting that would cause API clients to have to use tools like mechanize to fill in forms, which would be difficult.
>
> This authentication expects that you're already followed some sort of OAuth flow & that the credentials (Nonce/token/etc) are simply being passed to it. It merely checks that the credentials are valid. No requests are made to remote services as part of this authentication class.

MultiAuthentication

This authentication class actually wraps any number of other authentication classes, attempting each until successfully authenticating. For example:

```
from django.contrib.auth.models import User
from tastypie.authentication import BasicAuthentication, ApiKeyAuthentication,␣
↪MultiAuthentication
from tastypie.authorization import DjangoAuthorization
from tastypie.resources import ModelResource

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

        authentication = MultiAuthentication(BasicAuthentication(),␣
↪ApiKeyAuthentication())
        authorization = DjangoAuthorization()
```

In the case of an authentication returning a customized HttpUnauthorized, MultiAuthentication defaults to the first returned one. Authentication schemes that need to control the response, such as the included BasicAuthentication and DigestAuthentication, should be placed first.

Implementing Your Own Authentication/Authorization

Implementing your own `Authentication` classes is a simple process. `Authentication` has two methods to override (one of which is optional but recommended to be customized):

```
from tastypie.authentication import Authentication


class SillyAuthentication(Authentication):
    def is_authenticated(self, request, **kwargs):
        if 'daniel' in request.user.username:
          return True

        return False

    # Optional but recommended
    def get_identifier(self, request):
        return request.user.username
```

Under this scheme, only users with 'daniel' in their username will be allowed in.

CHAPTER 16

Authorization

Authorization is the component needed to verify what someone can do with the resources within an API.

Authorization answers the question "Is permission granted for this user to take this action?" This usually involves checking permissions such as Create/Read/Update/Delete access, or putting limits on what data the user can access.

Usage

Using these classes is simple. Simply provide them (or your own class) as a `Meta` option to the `Resource` in question. For example:

```
from django.contrib.auth.models import User
from tastypie.authorization import DjangoAuthorization
from tastypie.resources import ModelResource


class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
        resource_name = 'auth/user'
        excludes = ['email', 'password', 'is_superuser']
        # Add it here.
        authorization = DjangoAuthorization()
```

Authorization Options

Tastypie ships with the following `Authorization` classes:

Authorization

The no-op authorization option, no permissions checks are performed.

> **Warning:** This is a potentially dangerous option, as it means *ANY* recognized user can modify *ANY* data they encounter in the API. Be careful who you trust.

ReadOnlyAuthorization

This authorization class only permits reading data, regardless of what the `Resource` might think is allowed. This is the default `Authorization` class and the safe option.

DjangoAuthorization

The most advanced form of authorization, this checks the permission a user has granted to them on the resource's model (via `django.contrib.auth.models.Permission`). In conjunction with the admin, this is a very effective means of control.

The permissions required using `DjangoAuthorization` follow Django Admin's implementation and are as follows:

HTTP + URI	Method	User's permissions required to grant access	Includes check for
`POST <resource>/`	create_list	`add`	
`POST <resource>/<id>` (*)	create_detail	`add`	
`GET <resource>/`	read_list	`change`	
`GET <resouce>/<id>`	read_detail	`change`	
`PUT <resource>/`	update_list	`change`	`read_list`
`PUT <resource>/<id>`	update_detail	`change`	`read_detail`
`DELETE <resource>/`	delete_list	`delete`	`read_list`
`DELETE <resource>/`	delete_detail	`delete`	`read_detail`

(*) The permission check for `create_detail` is implemented in `DjangoAuthorization`, however ModelResource does not provide an implementation and raises HttpNotImplemented.

Notes:

- The actual permission checked is *<app_label>.<permission>_<model>* where app_label is derived from the resource's model (e.g. *myapp.change_foomodel*)
- *PUT* may revert to *POST* behavior and create new object(s) if the object(s) are not found. In this case the respective *create* permissions are required, instead of the usual *update* permissions.
- Requiring *change* for both read and update is such to keep consistent with Django Admin. To override this behavior and require a custom permission, override DjangoAuthorization as follows:

```
class CustomDjangoAuthorization(DjangoAuthorization):
    READ_PERM_CODE = 'view` # matching respective Permission.codename
```

The `Authorization` API

An `Authorization`-compatible class implements the following methods:

- `read_list`
- `read_detail`
- `create_list`
- `create_detail`
- `update_list`
- `update_detail`
- `delete_list`
- `delete_detail`

Each method takes two parameters, `object_list` & `bundle`.

`object_list` is the collection of objects being processed as part of the request. **FILTERING** & other restrictions to the set will have already been applied prior to this call.

`bundle` is the populated `Bundle` object for the request. You'll likely frequently be accessing `bundle.request.user`, though raw access to the data can be helpful.

What you return from the method varies based on the type of method.

Return Values: The List Case

In the case of the `*_list` methods, you'll want to filter the `object_list` & return only the objects the user has access to.

Returning an empty list simply won't allow the action to be applied to any objects. However, they will not get a HTTP error status code.

If you'd rather they received an unauthorized status code, raising `Unauthorized` will return a HTTP `401`.

Return Values: The Detail Case

In the case of the `*_detail` methods, you'll have access to the `object_list` (so you know if a given object fits within the overall set), **BUT** you'll want to be inspecting `bundle.obj` & either returning `True` if they should be allowed to continue or raising the `Unauthorized` exception if not.

Raising `Unauthorized` will cause a HTTP `401` error status code in the response.

Implementing Your Own Authorization

Implementing your own `Authorization` classes is a relatively simple process. Anything that is API-compatible is acceptable, only the method names matter to Tastypie.

An example implementation of a user only being able to access or modify "their" objects might look like:

```
from tastypie.authorization import Authorization
from tastypie.exceptions import Unauthorized


class UserObjectsOnlyAuthorization(Authorization):
    def read_list(self, object_list, bundle):
        # This assumes a ``QuerySet`` from ``ModelResource``.
        return object_list.filter(user=bundle.request.user)

    def read_detail(self, object_list, bundle):
        # Is the requested object owned by the user?
        return bundle.obj.user == bundle.request.user

    def create_list(self, object_list, bundle):
        # Assuming they're auto-assigned to ``user``.
        return object_list

    def create_detail(self, object_list, bundle):
        return bundle.obj.user == bundle.request.user

    def update_list(self, object_list, bundle):
        allowed = []

        # Since they may not all be saved, iterate over them.
        for obj in object_list:
            if obj.user == bundle.request.user:
                allowed.append(obj)

        return allowed

    def update_detail(self, object_list, bundle):
        return bundle.obj.user == bundle.request.user

    def delete_list(self, object_list, bundle):
        # Sorry user, no deletes for you!
        raise Unauthorized("Sorry, no deletes.")

    def delete_detail(self, object_list, bundle):
        raise Unauthorized("Sorry, no deletes.")
```

CHAPTER 17

Serialization

Serialization can be one of the most contentious areas of an API. Everyone has their own requirements, their own preferred output format & the desire to have control over what is returned.

As a result, Tastypie ships with a serializer that tries to meet the basic needs of most use cases, and the flexibility to go outside of that when you need to.

Usage

Using this class is simple. It is the default option on all `Resource` classes unless otherwise specified. The following code is identical to the defaults but demonstrate how you could use your own serializer:

```
from django.contrib.auth.models import User
from tastypie.resources import ModelResource
from tastypie.serializers import Serializer


class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
        resource_name = 'auth/user'
        excludes = ['email', 'password', 'is_superuser']
        # Add it here.
        serializer = Serializer()
```

Configuring Allowed Formats

The default `Serializer` supports the following formats:

- json
- jsonp (Disabled by default)

- xml
- yaml
- plist (see http://explorapp.com/biplist/)

Not everyone wants to install or support all the serialization options. If you would like to customize the list of supported formats for your entire site the *TASTYPIE_DEFAULT_FORMATS setting* allows you to set the default format list site-wide.

If you wish to change the format list for a specific resource, you can pass the list of supported formats using the `formats=` kwarg. For example, to provide only JSON & binary plist serialization:

```
from django.contrib.auth.models import User
from tastypie.resources import ModelResource
from tastypie.serializers import Serializer


class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
        resource_name = 'auth/user'
        excludes = ['email', 'password', 'is_superuser']
        serializer = Serializer(formats=['json', 'plist'])
```

Enabling the built-in (but disabled by default) JSONP support looks like:

```
from django.contrib.auth.models import User
from tastypie.resources import ModelResource
from tastypie.serializers import Serializer


class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
        resource_name = 'auth/user'
        excludes = ['email', 'password', 'is_superuser']
        serializer = Serializer(formats=['json', 'jsonp', 'xml', 'yaml', 'plist'])
```

Serialization Security

Deserialization of input from unknown or untrusted sources is an intrinsically risky endeavor and vulnerabilities are regularly found in popular format libraries. Tastypie adopts and recommends the following approach:

- Support the minimum required set of formats in your application. If you do not require a format, it's much safer to disable it completely. See *TASTYPIE_DEFAULT_FORMATS setting*.
- Some parsers offer additional safety check for use with untrusted content. The standard Tastypie Serializer attempts to be secure by default using features like PyYAML's safe_load function and the defusedxml security wrapper for popular Python XML libraries.

Note: Tastypie's precautions only apply to the default `Serializer`. If you have written your own serializer subclass we strongly recommend that you review your code to ensure that it uses the same precautions.

If backwards compatibility forces you to load files which require risky features we strongly recommend enabling those features only for the necessary resources and making your authorization checks as strict as possible. The *Authentication* and *Authorization* checks happen before deserialization so, for example, a resource which only

allowed POST or PUT requests to be made by administrators is far less exposed than a general API open to the unauthenticated internet.

Implementing Your Own Serializer

There are several different use cases here. We'll cover simple examples of wanting a tweaked format & adding a different format.

To tweak a format, simply override it's `to_<format>` & `from_<format>` methods. So adding the server time to all output might look like so:

```
import time
import json
from django.core.serializers.json import DjangoJSONEncoder
from tastypie.serializers import Serializer

class CustomJSONSerializer(Serializer):
    def to_json(self, data, options=None):
        options = options or {}

        data = self.to_simple(data, options)

        # Add in the current time.
        data['requested_time'] = time.time()

        return json.dumps(data, cls=DjangoJSONEncoder, sort_keys=True)

    def from_json(self, content):
        data = json.loads(content)

        if 'requested_time' in data:
            # Log the request here...
            pass

        return data
```

In the case of adding a different format, let's say you want to add a CSV output option to the existing set. Your `Serializer` subclass might look like:

```
import csv
import StringIO
from tastypie.serializers import Serializer


class CSVSerializer(Serializer):
    formats = Serializer.formats + ['csv']

    content_types = dict(
        Serializer.content_types.items() +
        [('csv', 'text/csv')])

    def to_csv(self, data, options=None):
        options = options or {}
        data = self.to_simple(data, options)
        raw_data = StringIO.StringIO()
```

```
        if data['objects']:
            fields = data['objects'][0].keys()
            writer = csv.DictWriter(raw_data, fields,
                                    dialect="excel",
                                    extrasaction='ignore')
            header = dict(zip(fields, fields))
            writer.writerow(header)  # In Python 2.7: `writer.writeheader()`
            for item in data['objects']:
                writer.writerow(item)

        return raw_data.getvalue()

    def from_csv(self, content):
        raw_data = StringIO.StringIO(content)
        data = []
        # Untested, so this might not work exactly right.
        for item in csv.DictReader(raw_data):
            data.append(item)
        return data
```

`Serializer` Methods

A swappable class for serialization.

This handles most types of data as well as the following output formats:

```
* json
* jsonp
* xml
* yaml
* plist
```

It was designed to make changing behavior easy, either by overridding the various format methods (i.e. `to_json`), by changing the `formats/content_types` options or by altering the other hook methods.

`get_mime_for_format`

`Serializer.get_mime_for_format(self, format):`

Given a format, attempts to determine the correct MIME type.

If not available on the current `Serializer`, returns `application/json` by default.

`format_datetime`

`Serializer.format_datetime(data):`

A hook to control how datetimes are formatted.

Can be overridden at the `Serializer` level (`datetime_formatting`) or globally (via `settings.TASTYPIE_DATETIME_FORMATTING`).

Default is `iso-8601`, which looks like "2010-12-16T03:02:14".

format_date

Serializer.format_date(data):

A hook to control how dates are formatted.

Can be overridden at the `Serializer` level (`datetime_formatting`) or globally (via `settings.TASTYPIE_DATETIME_FORMATTING`).

Default is `iso-8601`, which looks like "2010-12-16".

format_time

Serializer.format_time(data):

A hook to control how times are formatted.

Can be overridden at the `Serializer` level (`datetime_formatting`) or globally (via `settings.TASTYPIE_DATETIME_FORMATTING`).

Default is `iso-8601`, which looks like "03:02:14".

serialize

Serializer.serialize(self, bundle, format='application/json', options={}):

Given some data and a format, calls the correct method to serialize the data and returns the result.

deserialize

Serializer.deserialize(self, content, format='application/json'):

Given some data and a format, calls the correct method to deserialize the data and returns the result.

to_simple

Serializer.to_simple(self, data, options):

For a piece of data, attempts to recognize it and provide a simplified form of something complex.

This brings complex Python data structures down to native types of the serialization format(s).

to_etree

Serializer.to_etree(self, data, options=None, name=None, depth=0):

Given some data, converts that data to an `etree.Element` suitable for use in the XML output.

from_etree

Serializer.from_etree(self, data):

Not the smartest deserializer on the planet. At the request level, it first tries to output the deserialized subelement called "object" or "objects" and falls back to deserializing based on hinted types in the XML element attribute "type".

to_json

Serializer.to_json(self, data, options=None):

Given some Python data, produces JSON output.

from_json

Serializer.from_json(self, content):

Given some JSON data, returns a Python dictionary of the decoded data.

to_jsonp

Serializer.to_jsonp(self, data, options=None):

Given some Python data, produces JSON output wrapped in the provided callback.

to_xml

Serializer.to_xml(self, data, options=None):

Given some Python data, produces XML output.

from_xml

Serializer.from_xml(self, content):

Given some XML data, returns a Python dictionary of the decoded data.

to_yaml

Serializer.to_yaml(self, data, options=None):

Given some Python data, produces YAML output.

from_yaml

Serializer.from_yaml(self, content):

Given some YAML data, returns a Python dictionary of the decoded data.

to_plist

Serializer.to_plist(self, data, options=None):

Given some Python data, produces binary plist output.

from_plist

Serializer.from_plist(self, content):

Given some binary plist data, returns a Python dictionary of the decoded data.

CHAPTER 18

Throttling

Sometimes, the client on the other end may request data too frequently or you have a business use case that dictates that the client should be limited to a certain number of requests per hour.

For this, Tastypie includes throttling as a way to limit the number of requests in a timeframe.

Usage

To specify a throttle, add the `Throttle` class to the `Meta` class on the `Resource`:

```
from django.contrib.auth.models import User
from tastypie.resources import ModelResource
from tastypie.throttle import BaseThrottle


class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
        resource_name = 'auth/user'
        excludes = ['email', 'password', 'is_superuser']
        # Add it here.
        throttle = BaseThrottle(throttle_at=100)
```

Throttle Options

Each of the `Throttle` classes accepts the following initialization arguments:

- `throttle_at` - the number of requests at which the user should be throttled. Default is 150 requests.
- `timeframe` - the length of time (in seconds) in which the user make up to the `throttle_at` requests. Default is 3600 seconds (1 hour).

- `expiration` - the length of time to retain the times the user has accessed the api in the cache. Default is 604800 (1 week).

Tastypie ships with the following `Throttle` classes:

BaseThrottle

The no-op throttle option, this does no throttling but implements much of the common logic and serves as an api-compatible plug. Very useful for development.

CacheThrottle

This uses just the cache to manage throttling. Fast but prone to cache misses and/or cache restarts.

CacheDBThrottle

A write-through option that uses the cache first & foremost, but also writes through to the database to persist access times. Useful for logging client accesses & with RAM-only caches.

Implementing Your Own Throttle

Writing a `Throttle` class is not quite as simple as the other components. There are two important methods, `should_be_throttled` & `accessed`. The `should_be_throttled` method dictates whether or not the client should be throttled. The `accessed` method allows for the recording of the hit to the API.

An example of a subclass might be:

```
import random
from tastypie.throttle import BaseThrottle


class RandomThrottle(BaseThrottle):
    def should_be_throttled(self, identifier, **kwargs):
        if random.randint(0, 10) % 2 == 0:
          return True

        return False

    def accessed(self, identifier, **kwargs):
        pass
```

This throttle class would pick a random number between 0 & 10. If the number is even, their request is allowed through; otherwise, their request is throttled & rejected.

Usage with Resource

Using throttling with something like search, requires that you call `throttle_check` and `log_throttled_access` explicitly.

An example of this might be:

```
from tastypie.throttle import CacheThrottle


class NoteResource(Resource):
    class Meta:
        allowed_methods = ['get']
        resource_name = 'notes'
        throttle = CacheThrottle()

    def prepend_urls(self):
        return [
            url(r"^(?P<resource_name>%s)/search%s$" % (self._meta.resource_name,␣
↪trailing_slash()), self.wrap_view('get_search'), name="api_get_search"),
        ]

    def search(self, request, **kwargs):
        self.method_check(request, allowed=self.Meta.allowed_methods)
        self.is_authenticated(request)
        self.throttle_check(request)
        self.log_throttled_access(request)

        # Do the query.
        sqs = SearchQuerySet().models(Note).load_all().auto_query(request.GET.get('q',
↪ ''))
        paginator = Paginator(sqs, 20)

        try:
            page = paginator.page(int(request.GET.get('page', 1)))
        except InvalidPage:
            raise Http404("Sorry, no results on that page.")

        objects = []

        for result in page.object_list:
            bundle = self.build_bundle(obj=result.object, request=request)
            bundle = self.full_dehydrate(bundle)
            objects.append(bundle)

        object_list = {
            'objects': objects,
        }

        return self.create_response(request, object_list)
```

CHAPTER 19

Paginator

Similar to Django's `Paginator`, Tastypie includes a `Paginator` object which limits result sets down to sane amounts for passing to the client.

This is used in place of Django's `Paginator` due to the way pagination works. `limit` & `offset` (tastypie) are used in place of `page` (Django) so none of the page-related calculations are necessary.

This implementation also provides additional details like the `total_count` of resources seen and convenience links to the `previous`/`next` pages of data as available.

Usage

Using this class is simple, but slightly different than the other classes used by Tastypie. Like the others, you provide the `Paginator` (or your own subclass) as a `Meta` option to the `Resource` in question. **Unlike** the others, you provide the class, *NOT* an instance. For example:

```
from django.contrib.auth.models import User
from tastypie.paginator import Paginator
from tastypie.resources import ModelResource


class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
        resource_name = 'auth/user'
        excludes = ['email', 'password', 'is_superuser']
        # Add it here.
        paginator_class = Paginator
```

Warning: The default paginator contains the `total_count` value, which shows how many objects are in the underlying object list.

Obtaining this data from the database may be inefficient, especially with large datasets, and unfiltered API requests.

See http://wiki.postgresql.org/wiki/Slow_Counting and http://www.wikivs.com/wiki/MySQL_vs_PostgreSQL#COUNT.28.2A.29 for reference, on why this may be a problem when using PostgreSQL and MySQL's InnoDB engine.

Here's an *example solution* to this problem.

Implementing Your Own Paginator

Adding other features to a paginator usually consists of overriding one of the built-in methods. For instance, adding a page number to the output might look like:

```
from tastypie.paginator import Paginator


class PageNumberPaginator(Paginator):
    def page(self):
        output = super(PageNumberPaginator, self).page()
        output['page_number'] = int(self.offset / self.limit) + 1
        return output
```

Another common request is to alter the structure Tastypie uses in the list view. Here's an example of renaming:

```
from tastypie.paginator import Paginator


class BlogEntryPaginator(Paginator):
    def page(self):
        output = super(BlogEntryPaginator, self).page()

        # First keep a reference.
        output['pagination'] = output['meta']
        output['entries'] = output['objects']

        # Now nuke the original keys.
        del output['meta']
        del output['objects']

        return output
```

Estimated count instead of total count

Here's an example, of how you can omit `total_count` from the resource, and instead add an `estimated_count` for efficiency. See the warning above for details:

```
import json

from django.db import connection

from tastypie.paginator import Paginator


```

```
class EstimatedCountPaginator(Paginator):

    def get_next(self, limit, offset, count):
        # The parent method needs an int which is higher than "limit + offset"
        # to return a url. Setting it to an unreasonably large value, so that
        # the parent method will always return the url.
        count = 2 ** 64
        return super(EstimatedCountPaginator, self).get_next(limit, offset, count)

    def get_count(self):
        return None

    def get_estimated_count(self):
        """Get the estimated count by using the database query planner."""
        # If you do not have PostgreSQL as your DB backend, alter this method
        # accordingly.
        return self._get_postgres_estimated_count()

    def _get_postgres_estimated_count(self):

        # This method only works with postgres >= 9.0.
        # If you need postgres vesrions less than 9.0, remove "(format json)"
        # below and parse the text explain output.

        def _get_postgres_version():
            # Due to django connections being lazy, we need a cursor to make
            # sure the connection.connection attribute is not None.
            connection.cursor()
            return connection.connection.server_version

        try:
            if _get_postgres_version() < 90000:
                return
        except AttributeError:
            return

        cursor = connection.cursor()
        query = self.objects.all().query

        # Remove limit and offset from the query, and extract sql and params.
        query.low_mark = None
        query.high_mark = None
        query, params = self.objects.query.sql_with_params()

        # Fetch the estimated rowcount from EXPLAIN json output.
        query = 'explain (format json) %s' % query
        cursor.execute(query, params)
        explain = cursor.fetchone()[0]
        # Older psycopg2 versions do not convert json automatically.
        if isinstance(explain, basestring):
            explain = json.loads(explain)
        rows = explain[0]['Plan']['Plan Rows']
        return rows

    def page(self):
        data = super(EstimatedCountPaginator, self).page()
        data['meta']['estimated_count'] = self.get_estimated_count()
        return data
```

CHAPTER 20

GeoDjango

Tastypie features support for GeoDjango! Resources return and accept GeoJSON (or similarly-formatted analogs for other formats) and all spatial lookup filters are supported. Distance lookups are not yet supported.

Usage

Here's an example geographic model for leaving notes in polygonal regions:

```
from django.contrib.gis import models

class GeoNote(models.Model):
    content = models.TextField()
    polys = models.MultiPolygonField(null=True, blank=True)

    objects = models.GeoManager()
```

To define a resource that takes advantage of the geospatial features, we use `tastypie.contrib.gis.resources.ModelResource`:

```
from tastypie.contrib.gis.resources import ModelResource

class GeoNoteResource(ModelResource):
    class Meta:
        resource_name = 'geonotes'
        queryset = GeoNote.objects.all()

        filtering = {
            'polys': ALL,
        }
```

Now when we do a `GET` on our GeoNoteResource we get back GeoJSON in our response:

```
{
    "content": "My note content",
    "id": "1",
    "polys": {
        "coordinates": [[[
            [-122.511067, 37.771276], [-122.510037, 37.766390999999999],
            [-122.510037, 37.763812999999999], [-122.456822, 37.765847999999998],
            [-122.45296, 37.766458999999998], [-122.454848, 37.773989999999998],
            [-122.475362, 37.773040000000002], [-122.511067, 37.771276]
        ]]],
        "type": "MultiPolygon"
    },
    "resource_uri": "/api/v1/geonotes/1/"
}
```

When updating or creating new resources, simply provide GeoJSON or the GeoJSON analog for your perferred format.

Filtering

We can filter using any standard GeoDjango spatial lookup filter. Simply provide a GeoJSON (or the analog) as a `GET` parameter value.

Let's find all of our `GeoNote` resources that contain a point inside of Golden Gate Park:

```
/api/v1/geonotes/?polys__contains={"type": "Point", "coordinates": [-122.475233, 37.
↪768617]}
```

Returns:

```
{
    "meta": {
        "limit": 20, "next": null, "offset": 0, "previous": null, "total_count": 1},
    "objects": [
        {
            "content": "My note content",
            "id": "1",
            "polys": {
                "coordinates": [[[
                    [-122.511067, 37.771276], [-122.510037, 37.766390999999999],
                    [-122.510037, 37.763812999999999], [-122.456822, 37.
↪765847999999998],
                    [-122.45296, 37.766458999999998], [-122.454848, 37.
↪773989999999998],
                    [-122.475362, 37.773040000000002], [-122.511067, 37.771276]
                ]]],
                "type": "MultiPolygon"
            },
            "resource_uri": "/api/geonotes/1/"
        }
    ]
}
```

We get back the `GeoNote` resource defining Golden Gate Park. Awesome!

CHAPTER 21

ContentTypes and GenericForeignKeys

Content Types and GenericForeignKeys are for relationships where the model on one end is not defined by the model's schema.

If you're using GenericForeignKeys in django, you can use a GenericForeignKeyField in Tastypie.

Usage

Here's an example model with a GenericForeignKey taken from the Django docs:

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

class TaggedItem(models.Model):
    tag = models.SlugField()
    content_type = models.ForeignKey(ContentType, on_delete=models.CASCADE)
    object_id = models.PositiveIntegerField()
    content_object = generic.GenericForeignKey('content_type', 'object_id')

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

A simple ModelResource for this model might look like this:

```
from tastypie.contrib.contenttypes.fields import GenericForeignKeyField
from tastypie.resources import ModelResource

from .models import Note, Quote, TaggedItem


class QuoteResource(ModelResource):

    class Meta:
```

```
        resource_name = 'quotes'
        queryset = Quote.objects.all()


class NoteResource(ModelResource):

    class Meta:
        resource_name = 'notes'
        queryset = Note.objects.all()


class TaggedItemResource(ModelResource):
    content_object = GenericForeignKeyField({
        Note: NoteResource,
        Quote: QuoteResource
    }, 'content_object')

    class Meta:
        resource_name = 'tagged_items'
        queryset = TaggedItem.objects.all()
```

A ModelResource that is to be used as a relation to a GenericForeignKeyField must also be registered to the Api instance defined in your URLconf in order to provide a reverse uri for lookups.

Like ToOneField, you must add your GenericForeignKey attribute to your ModelResource definition. It will not be added automatically like most other field or attribute types. When you define it, you must also define the other models and match them to their resources in a dictionary, and pass that as the first argument, the second argument is the name of the attribute on the model that holds the GenericForeignKey.

CHAPTER 22

Namespaces

For various reasons you might want to deploy your API under a namespaced URL path. To support that tastypie includes `NamespacedApi` and `NamespacedModelResource`.

A sample definition of your API in this case would be something like:

```
from django.conf.urls import url, include
from tastypie.api import NamespacedApi
from my_application.api.resources import NamespacedUserResource

api = NamespacedApi(api_name='v1', urlconf_namespace='special')
api.register(NamespacedUserResource())

urlpatterns = [
    url(r'^api/', include(api.urls, namespace='special')),
]
```

And your model resource:

```
from django.contrib.auth.models import User
from tastypie.resources import NamespacedModelResource
from tastypie.authorization import Authorization

class NamespacedUserResource(NamespacedModelResource):
    class Meta:
        resource_name = 'users'
        queryset = User.objects.all()
        authorization = Authorization()
```

CHAPTER 23

Tastypie Cookbook

Creating a Full OAuth 2.0 API

It is common to use django to provision OAuth 2.0 tokens for users and then have Tasty Pie use these tokens to authenticate users to the API. Follow this tutorial and use this custom authentication class to enable OAuth 2.0 authentication with Tasty Pie.

```
# api.py
from tastypie import fields
from tastypie.authorization import DjangoAuthorization
from tastypie.resources import ModelResource, Resource
from myapp.models import Poll, Choice
from authentication import OAuth20Authentication


class ChoiceResource(ModelResource):
    class Meta:
        queryset = Choice.objects.all()
        resource_name = 'choice'
        authorization = DjangoAuthorization()
        authentication = OAuth20Authentication()


class PollResource(ModelResource):
    choices = fields.ToManyField(ChoiceResource, 'choice_set', full=True)

    class Meta:
        queryset = Poll.objects.all()
        resource_name = 'poll'
        authorization = DjangoAuthorization()
        authentication = OAuth20Authentication()
```

Adding Custom Values

You might encounter cases where you wish to include additional data in a response which is not obtained from a field or method on your model. You can easily extend the `dehydrate()` method to provide additional values:

```
from myapp.models import MyModel


class MyModelResource(Resource):
    class Meta:
        queryset = MyModel.objects.all()

    def dehydrate(self, bundle):
        bundle.data['custom_field'] = "Whatever you want"
        return bundle
```

Per-Request Alterations To The Queryset

A common pattern is needing to limit a queryset by something that changes per-request, for instance the date/time. You can accomplish this by lightly modifying `get_object_list`:

```
from django.utils import timezone
from myapp.models import MyModel


class MyModelResource(ModelResource):
    class Meta:
        queryset = MyModel.objects.all()

    def get_object_list(self, request):
        return super(MyModelResource, self).get_object_list(request).filter(start_
↪date__gte=timezone.now())
```

Using Your `Resource` In Regular Views

In addition to using your resource classes to power the API, you can also use them to write other parts of your application, such as your views. For instance, if you wanted to encode user information in the page for some Javascript's use, you could do the following. In this case, `user_json` will not include a valid `resource_uri`:

```
# views.py
from django.shortcuts import render_to_response
from myapp.api.resources import UserResource


def user_detail(request, username):
    res = UserResource()
    request_bundle = res.build_bundle(request=request)
    user = res.obj_get(request_bundle, username=username)

    # Other things get prepped to go into the context then...

    user_bundle = res.build_bundle(request=request, obj=user)
```

```
    user_json = res.serialize(None, res.full_dehydrate(user_bundle), "application/json
↪")

    return render_to_response("myapp/user_detail.html", {
        # Other things here.
        "user_json": user_json,
    })
```

To include a valid `resource_uri`, the resource must be associated with an `tastypie.Api` instance, as below:

```
# urls.py
from tastypie.api import Api
from myapp.api.resources import UserResource


my_api = Api(api_name='v1')
my_api.register(UserResource())

# views.py
from myapp.urls import my_api


def user_detail(request, username):
    res = my_api.canonical_resource_for('user')
    # continue as above...
```

Alternatively, to get a valid `resource_uri` you may pass in the `api_name` parameter directly to the Resource:

```
# views.py
from django.shortcuts import render_to_response
from myapp.api.resources import UserResource


def user_detail(request, username):
    res = UserResource(api_name='v1')
    # continue as above...
```

Example of getting a list of users:

```
def user_list(request):
    res = UserResource()
    request_bundle = res.build_bundle(request=request)
    queryset = res.obj_get_list(request_bundle)

    bundles = []
    for obj in queryset:
        bundle = res.build_bundle(obj=obj, request=request)
        bundles.append(res.full_dehydrate(bundle, for_list=True))

    list_json = res.serialize(None, bundles, "application/json")

    return render_to_response('myapp/user_list.html', {
        # Other things here.
        "list_json": list_json,
    })
```

Then in template you could convert JSON into JavaScript object:

```
<script>
    var json = "{{ list_json|escapejs }}";
    var users = JSON.parse(json);
</script>
```

Using Non-PK Data For Your URLs

By convention, `ModelResources` usually expose the detail endpoints utilizing the primary key of the `Model` they represent. However, this is not a strict requirement. Each URL can take other named URLconf parameters that can be used for the lookup.

For example, if you want to expose `User` resources by username, you can do something like the following:

```
# myapp/api/resources.py
from django.conf.urls import url
from django.contrib.auth.models import User


class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
        detail_uri_name = 'username'

    def prepend_urls(self):
        return [
            url(r"^(?P<resource_name>%s)/(?P<username>[\w\d_.-]+)/$" % self._meta.
↪resource_name, self.wrap_view('dispatch_detail'), name="api_dispatch_detail"),
        ]
```

The added URLconf matches before the standard URLconf included by default & matches on the username provided in the URL.

Another alternative approach is to override the `dispatch` method:

```
# myapp/api/resources.py
from myapp.models import MyModel

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

    class Meta:
        queryset = MyModel.objects.all()
        resource_name = 'mymodel'

    def dispatch(self, request_type, request, **kwargs):
        username = kwargs.pop('username')
        kwargs['user'] = get_object_or_404(User, username=username)
        return super(MyModelResource, self).dispatch(request_type, request, **kwargs)

# urls.py
from django.conf.urls import url, include

mymodel_resource = MyModelResource()

urlpatterns = [
    # The normal jazz here, then...
```

```
    url(r'^api/(?P<username>\w+)/', include(mymodel_resource.urls)),
]
```

Nested Resources

You can also do "nested resources" (resources within another related resource) by lightly overriding the `prepend_urls` method & adding on a new method to handle the children:

```
class ChildResource(ModelResource):
    pass

from tastypie.utils import trailing_slash

class ParentResource(ModelResource):
    children = fields.ToManyField(ChildResource, 'children')

    def prepend_urls(self):
        return [
            url(r"^(?P<resource_name>%s)/(?P<pk>\w[\w/-]*)/children%s$" % (self._meta.
↪resource_name, trailing_slash()), self.wrap_view('get_children'), name="api_get_
↪children"),
        ]

    def get_children(self, request, **kwargs):
        try:
            bundle = self.build_bundle(data={'pk': kwargs['pk']}, request=request)
            obj = self.cached_obj_get(bundle=bundle, **self.remove_api_resource_
↪names(kwargs))
        except ObjectDoesNotExist:
            return HttpGone()
        except MultipleObjectsReturned:
            return HttpMultipleChoices("More than one resource is found at this URI.")

        child_resource = ChildResource()
        return child_resource.get_list(request, parent_id=obj.pk)
```

Adding Search Functionality

Another common request is being able to integrate search functionality. This approach uses Haystack, though you could hook it up to any search technology. We leave the CRUD methods of the resource alone, choosing to add a new endpoint at `/api/v1/notes/search/`:

```
from django.conf.urls import url, include
from django.core.paginator import Paginator, InvalidPage
from django.http import Http404
from haystack.query import SearchQuerySet
from tastypie.resources import ModelResource
from tastypie.utils import trailing_slash
from notes.models import Note


class NoteResource(ModelResource):
    class Meta:
```

```
        queryset = Note.objects.all()
        resource_name = 'notes'

    def prepend_urls(self):
        return [
            url(r"^(?P<resource_name>%s)/search%s$" % (self._meta.resource_name,␣
↪trailing_slash()), self.wrap_view('get_search'), name="api_get_search"),
        ]

    def get_search(self, request, **kwargs):
        self.method_check(request, allowed=['get'])
        self.is_authenticated(request)
        self.throttle_check(request)

        # Do the query.
        sqs = SearchQuerySet().models(Note).load_all().auto_query(request.GET.get('q',
↪ ''))
        paginator = self._meta.paginator_class(request.GET, sqs,
            resource_uri=self.get_resource_uri(), limit=self._meta.limit,
            max_limit=self._meta.max_limit, collection_name=self._meta.collection_
↪name)

        to_be_serialized = paginator.page()

        bundles = [self.build_bundle(obj=result.object, request=request) for result␣
↪in to_be_serialized['objects']]
        to_be_serialized['objects'] = [self.full_dehydrate(bundle) for bundle in␣
↪bundles]
        to_be_serialized = self.alter_list_data_to_serialize(request, to_be_
↪serialized)
        return self.create_response(request, to_be_serialized)
```

Creating per-user resources

One might want to create an API which will require every user to authenticate and every user will be working only with objects associated with them. Let's see how to implement it for two basic operations: listing and creation of an object.

For listing we want to list only objects for which 'user' field matches 'request.user'. This could be done by applying a filter in the `authorized_read_list` method of your resource.

For creating we'd have to wrap `obj_create` method of `ModelResource`. Then the resulting code will look something like:

```
# myapp/api/resources.py
from tastypie.authentication import ApiKeyAuthentication
from tastypie.authorization import Authorization


class MyModelResource(ModelResource):
    class Meta:
        queryset = MyModel.objects.all()
        resource_name = 'mymodel'
        list_allowed_methods = ['get', 'post']
        authentication = ApiKeyAuthentication()
        authorization = Authorization()
```

```
    def obj_create(self, bundle, **kwargs):
        return super(MyModelResource, self).obj_create(bundle, user=bundle.request.
↪user)

    def authorized_read_list(self, object_list, bundle):
        return object_list.filter(user=bundle.request.user)
```

camelCase JSON Serialization

The convention in the world of Javascript has standardized on camelCase, where Tastypie uses underscore syntax, which can lead to "ugly" looking code in Javascript. You can create a custom serializer that emits values in camelCase instead:

```
import re
import json
from tastypie.serializers import Serializer


class CamelCaseJSONSerializer(Serializer):
    formats = ['json']
    content_types = {
        'json': 'application/json',
    }

    def to_json(self, data, options=None):
        # Changes underscore_separated names to camelCase names to go from python␣
↪convention to javacsript convention
        data = self.to_simple(data, options)

        def underscoreToCamel(match):
            return match.group()[0] + match.group()[2].upper()

        def camelize(data):
            if isinstance(data, dict):
                new_dict = {}
                for key, value in data.items():
                    new_key = re.sub(r"[a-z]_[a-z]", underscoreToCamel, key)
                    new_dict[new_key] = camelize(value)
                return new_dict
            if isinstance(data, (list, tuple)):
                for i in range(len(data)):
                    data[i] = camelize(data[i])
                return data
            return data

        camelized_data = camelize(data)

        return json.dumps(camelized_data, sort_keys=True)

    def from_json(self, content):
        # Changes camelCase names to underscore_separated names to go from javascript␣
↪convention to python convention
        data = json.loads(content)
```

```
        def camelToUnderscore(match):
            return match.group()[0] + "_" + match.group()[1].lower()

        def underscorize(data):
            if isinstance(data, dict):
                new_dict = {}
                for key, value in data.items():
                    new_key = re.sub(r"[a-z][A-Z]", camelToUnderscore, key)
                    new_dict[new_key] = underscorize(value)
                return new_dict
            if isinstance(data, (list, tuple)):
                for i in range(len(data)):
                    data[i] = underscorize(data[i])
                return data
            return data

        underscored_data = underscorize(data)

        return underscored_data
```

Pretty-printed JSON Serialization

By default, Tastypie outputs JSON with no indentation or newlines (equivalent to calling `json.dumps()` with *indent* set to `None`). You can override this behavior in a custom serializer:

```
import json
from django.core.serializers.json import DjangoJSONEncoder
from tastypie.serializers import Serializer

class PrettyJSONSerializer(Serializer):
    json_indent = 2

    def to_json(self, data, options=None):
        options = options or {}
        data = self.to_simple(data, options)
        return json.dumps(data, cls=DjangoJSONEncoder,
                sort_keys=True, ensure_ascii=False, indent=self.json_indent)
```

Determining format via URL

Sometimes it's required to allow selecting the response format by specifying it in the API URL, for example `/api/v1/users.json` instead of `/api/v1/users/?format=json`. The following snippet allows that kind of syntax additional to the default URL scheme:

```
# myapp/api/resources.py

from django.contrib.auth.models import User
# Piggy-back on internal csrf_exempt existence handling
from tastypie.resources import csrf_exempt

class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
```

```
    def prepend_urls(self):
        """
        Returns a URL scheme based on the default scheme to specify
        the response format as a file extension, e.g. /api/v1/users.json
        """
        return [
            url(r"^(?P<resource_name>%s)\.(?P<format>\w+)$" % self._meta.resource_
↪name, self.wrap_view('dispatch_list'), name="api_dispatch_list"),
            url(r"^(?P<resource_name>%s)/schema\.(?P<format>\w+)$" % self._meta.
↪resource_name, self.wrap_view('get_schema'), name="api_get_schema"),
            url(r"^(?P<resource_name>%s)/set/(?P<pk_list>\w[\w/;-]*)\.(?P<format>\w+)$
↪" % self._meta.resource_name, self.wrap_view('get_multiple'), name="api_get_multiple
↪"),
            url(r"^(?P<resource_name>%s)/(?P<pk>\w[\w/-]*)\.(?P<format>\w+)$" % self._
↪meta.resource_name, self.wrap_view('dispatch_detail'), name="api_dispatch_detail"),
        ]

    def determine_format(self, request):
        """
        Used to determine the desired format from the request.format
        attribute.
        """
        if (hasattr(request, 'format') and
                request.format in self._meta.serializer.formats):
            return self._meta.serializer.get_mime_for_format(request.format)
        return super(UserResource, self).determine_format(request)

    def wrap_view(self, view):
        @csrf_exempt
        def wrapper(request, *args, **kwargs):
            request.format = kwargs.pop('format', None)
            wrapped_view = super(UserResource, self).wrap_view(view)
            return wrapped_view(request, *args, **kwargs)
        return wrapper
```

Adding to the Django Admin

If you're using the django admin and ApiKeyAuthentication, you may want to see or edit ApiKeys next to users. To do this, you need to unregister the built-in UserAdmin, alter the inlines, and re-register it. This could go in any of your admin.py files. You may also want to register ApiAccess and ApiKey models on their own.:

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

from tastypie.admin import ApiKeyInline


class UserModelAdmin(UserAdmin):
    inlines = UserAdmin.inlines + [ApiKeyInline]


admin.site.unregister(User)
admin.site.register(User, UserModelAdmin)
```

Using `SessionAuthentication`

If your users are logged into the site & you want Javascript to be able to access the API (assuming jQuery), the first thing to do is setup `SessionAuthentication`:

```
from django.contrib.auth.models import User
from tastypie.authentication import SessionAuthentication
from tastypie.resources import ModelResource


class UserResource(ModelResource):
    class Meta:
        resource_name = 'users'
        queryset = User.objects.all()
        authentication = SessionAuthentication()
```

Then you'd build a template like:

```
<html>
    <head>
        <title></title>
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js
↪"></script>
        <script type="text/javascript">
            $(document).ready(function() {
                // We use ``.ajax`` here due to the overrides.
                $.ajax({
                    // Substitute in your API endpoint here.
                    url: '/api/v1/users/',
                    contentType: 'application/json',
                    // The ``X-CSRFToken`` evidently can't be set in the
                    // ``headers`` option, so force it here.
                    // This method requires jQuery 1.5+.
                    beforeSend: function(jqXHR, settings) {
                        // Pull the token out of the DOM.
                        jqXHR.setRequestHeader('X-CSRFToken', $(
↪'input[name=csrfmiddlewaretoken]').val());
                    },
                    success: function(data, textStatus, jqXHR) {
                        // Your processing of the data here.
                        console.log(data);
                    }
                });
            });
        </script>
    </head>
    <body>
        <!-- Include the CSRF token in the body of the HTML -->
        {% csrf_token %}
    </body>
</html>
```

There are other ways to make this function, with other libraries or other techniques for supplying the token (see https://docs.djangoproject.com/en/dev/ref/contrib/csrf/#ajax for an alternative). This is simply a starting point for getting things working.

CHAPTER 24

Debugging Tastypie

There are some common problems people run into when using Tastypie for the first time. Some of the common problems and things to try appear below.

"I'm getting XML output in my browser but I want JSON output!"

This is actually not a bug and JSON support is present in your `Resource`. This issue is that Tastypie respects the `Accept` header your browser sends. Most browsers send something like:

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
```

Note that `application/xml` is the first format that Tastypie handles, hence you receive XML.

If you use `curl` from the command line, you should receive JSON by default:

```
curl http://localhost:8000/api/v1/
```

If you want JSON in the browser, simply append `?format=json` to your URL. Tastypie always respects this override first, before it falls back to the `Accept` header.

Querying using Tastypie's methods isn't working/returning multiple objects

When calling `obj_get` (or another method that uses it, such as `dispatch_detail`), mke sure the fields you're querying with are either `Meta.detail_uri_name` or a field which appears in `Meta.filtering`

"What's the format for a POST or PUT?"

You can view full schema for your resource through *Inspecting The Resource's Schema*.

In general, Tastypie will accept resources in the same format as it gives you. This means that you can see what any POST or PUT should look like by performing a GET of that resource.

Creating a duplicate of an entry, using Python and Requests:

```
import requests
import json

response = requests.get('http://localhost:8000/api/v1/entry/1/')
event = json.loads(response.content)

del event['id'] # We want the server to assign a new id

response = requests.post('http://localhost:8000/api/v1/entry/',
                         data=json.dumps(event),
                         headers={'content-type': 'application/json'})
```

The new event should be almost identical, with the exception of readonly fields. This method may fail if your model has a unique constraint, or otherwise fails validation.

This is less likely to happen on PUT, except for application logic changes (e.g. a *last_update* field). The following two `curl` commands replace and entry with an copy:

```
curl -H 'Accept: application/json' 'http://localhost:8000/api/v1/entry/1/' | \
curl -H 'Content-Type: application/json' -X PUT --data @- "http://localhost:8000/api/
↪v1/entry/1/"
```

You can do this over an entire collection as well:

```
curl -H 'Accept: application/json' 'http://localhost:8000/api/v1/entry/?limit=0' | \
curl -H 'Content-Type: application/json' -X PUT --data @- "http://localhost:8000/api/
↪v1/entry/"
```

CHAPTER 25

Sites Using Tastypie

The following sites are a partial list of sites using Tastypie. We're always interested in adding more sites, so please open a GitHub Issue or Pull Request for this page and we'll add you to the list.

LJWorld Marketplace

- http://www2.ljworld.com/marketplace/api/v1/?format=json

Forkinit

Read-only API access to recipes.

- http://forkinit.com/
- http://forkinit.com/api/v1/?format=json

Read The Docs

A hosted documentation site, primarily for Python docs. General purpose read-write access.

- http://readthedocs.org/
- http://readthedocs.org/api/v1/?format=json

Luzme

An e-book search site that lets you fetch pricing information.

- http://luzme.com/

- https://luzme.readthedocs.io/en/latest/

Politifact

To power their mobile (iPhone/Android/Playbook) applications.

- http://www.politifact.com/mobile/

LocalWiki

LocalWiki is a tool for collaborating in local, geographic communities. It's using Tastypie to provide an geospatially-aware REST API.

- https://localwiki.readthedocs.io/en/latest/api.html
- http://localwiki.org/blog/2012/aug/31/localwiki-api-released/

I-Am-CC.org

I-Am-CC.org is a tool for releasing Instagram photos under a Creative Commons license.

- http://i-am-cc.org/api/?format=json

Dbpatterns

Dbpatterns is a service that allows you to create, share, explore database models on the web.

- http://dbpatterns.com

CourtListener

Read-only API providing 2.5M legal opinions and other judicial data via Solr/Sunburnt and Postgres (Django models).

- Site: https://www.courtlistener.com
- Code : https://bitbucket.org/mlissner/search-and-awareness-platform-courtlistener/src

CHAPTER 26

Contributing

Tastypie is open-source and, as such, grows (or shrinks) & improves in part due to the community. Below are some guidelines on how to help with the project.

Philosophy

- Tastypie is BSD-licensed. All contributed code must be either
  - the original work of the author, contributed under the BSD, or...
  - work taken from another project released under a BSD-compatible license.
- GPL'd (or similar) works are not eligible for inclusion.
- Tastypie's git master branch should always be stable, production-ready & passing all tests.
- Major releases (1.x.x) are commitments to backward-compatibility of the public APIs. Any documented API should ideally not change between major releases. The exclusion to this rule is in the event of either a security issue or to accommodate changes in Django itself.
- Minor releases (x.3.x) are for the addition of substantial features or major bugfixes.
- Patch releases (x.x.4) are for minor features or bugfixes.

Guidelines For Reporting An Issue/Feature

So you've found a bug or have a great idea for a feature. Here's the steps you should take to help get it added/fixed in Tastypie:

- First, check to see if there's an existing issue/pull request for the bug/feature. All issues are at https://github.com/django-tastypie/django-tastypie/issues and pull reqs are at https://github.com/django-tastypie/django-tastypie/pulls.
- If there isn't one there, please file an issue. The ideal report includes:

  - A description of the problem/suggestion.
  - How to recreate the bug.
  - If relevant, including the versions of your:
    * Python interpreter
    * Django
    * Tastypie
    * Optionally of the other dependencies involved
  - Ideally, creating a pull request with a (failing) test case demonstrating what's wrong. This makes it easy for us to reproduce & fix the problem. Instructions for running the tests are at *Welcome to Tastypie!*

You might also hop into the IRC channel (`#tastypie` on `irc.freenode.net`) & raise your question there, as there may be someone who can help you with a work-around.

Guidelines For Contributing Code

If you're ready to take the plunge & contribute back some code/docs, the process should look like:

- Fork the project on GitHub into your own account.
- Clone your copy of Tastypie.
- Make a new branch in git & commit your changes there.
- Push your new branch up to GitHub.
- Again, ensure there isn't already an issue or pull request out there on it. If there is & you feel you have a better fix, please take note of the issue number & mention it in your pull request.
- Create a new pull request (based on your branch), including what the problem/feature is, versions of your software & referencing any related issues/pull requests.

In order to be merged into Tastypie, contributions must have the following:

- A solid patch that:
  - is clear.
  - works across all supported versions of Python/Django.
  - follows the existing style of the code base (mostly PEP-8).
  - comments included as needed.
- A test case that demonstrates the previous flaw that now passes with the included patch.
- If it adds/changes a public API, it must also include documentation for those changes.
- Must be appropriately licensed (see "Philosophy").
- Adds yourself to the AUTHORS file.

Please also:

- Unless your change only modifies the documentation, add the issue you're solving to the list in docs/release_notes/dev.rst, include issue and PR numbers.
- Squash your changes down to a single commit, or down to one commit containing your failing tests and one more commit containing the fix that makes those tests pass.

If your contribution lacks any of these things, they will have to be added by a core contributor before being merged into Tastypie proper, which may take substantial time for the all-volunteer team to get to.

Guidelines For Core Contributors

If you've been granted the commit bit, here's how to shepherd the changes in:

- Any time you go to work on Tastypie, please use `git pull --rebase` to fetch the latest changes.
- Any new features/bug fixes must meet the above guidelines for contributing code (solid patch/tests passing/docs included).
- Commits are typically cherry-picked onto a branch off master.
  - This is done so as not to include extraneous commits, as some people submit pull reqs based on their git master that has other things applied to it.
- A set of commits should be squashed down to a single commit.
  - `git merge --squash` is a good tool for performing this, as is `git rebase -i HEAD~N`.
  - This is done to prevent anyone using the git repo from accidently pulling work-in-progress commits.
- Commit messages should use past tense, describe what changed & thank anyone involved. Examples:

```
"""Added a new way to do per-object authorization."""
"""Fixed a bug in ``Serializer.to_xml``. Thanks to joeschmoe for the report!"""
"""BACKWARD-INCOMPATIBLE: Altered the arguments passed to ``Bundle.__init__``.

Further description appears here if the change warrants an explanation
as to why it was done."""
```

- For any patches applied from a contributor, please ensure their name appears in the AUTHORS file.
- When closing issues or pull requests, please reference the SHA in the closing message (i.e. `Thanks! Fixed in SHA: 6b93f6`). GitHub will automatically link to it.

CHAPTER 27

Release Notes

dev

The current in-progress version. Put your notes here so they can be easily copied to the release notes for the next release.

Bugfixes

- Example Bugfix (Closes #PR_Number)

v0.14.0

date 2017-07-03

Dropped support for all non-LTS versions of Django. Specifically supports Django 1.8 and 1.11.

Bugfixes

- Change OAuthAuthentication to use storage method to get user. (Closes #657)
- Fixed UnicodeDecodeError in _handle_500(). (Fixes #1190)
- Fix get_via_uri not working for alphabetic ids that contain the resource name (Fixes #1239, Closes #1240)
- Don't enable unsupported formats by default. (Fixes #1451)
- Gave ApiKey a __str__ implementation that works in Python 2 and 3. (Fixes #1459, Closes #1460)
- Improved admin UI for API Keys (Closes #1262)
- Avoid double query on *the_m2ms* in *ToManyField.dehydrate*. (Closes #433)

- Allow *ModelResource.Meta.fields = []* to disable introspection. *ModelResource.Meta.fields = None* or omitting *ModelResource.Meta.fields* allows introspection as usual. (Fixes #793)
- Added *Resource.get_response_class_for_exception* hook. (Closes #1154)
- Added UnsupportedSerializationFormat and UnsupportedDeserializationFormat exceptions, which are caught and result in HttpNotAcceptable (406 status) and HttpUnsupportedMediaType (415 status) responses, respectively. Previously these same types of errors woud have appeared as 400 BadRequest errors.
- Fix for datetime parsing error. (#1478)
- Gets rid of RemovedInDjango20Warning warning in Django 1.9 (Closes #1507)

v0.13.3

date 2016-02-17

This is the final release of Tastypie that is compatible with Django 1.9.

Bugfixes

- Permit changing existing value on a ToOneField to None. (Closes #1449)

v0.13.2

date 2016-02-14

Bugfixes

- Fix in *Resource.save_related*: *related_obj* can be empty in patch requests (introduced in #1378). (Fixes #1436)
- Fixed bug that prevented fitlering on related resources. *apply_filters* hook now used in obj_get. (Fixes #1435, Fixes #1443)
- Use *build_filters* in *obj_get*. (Fixes #1444)
- Updated DjangoAuthorization to disallow read unless a user has *change* permission. (#1407, PR #1409)
- Authorization classes now handle usernames containing spaces. Closes #966.
- **Cleaned up old, unneeded code. (closes PR #1433)**
  - Reuse Django test *Client.patch()*. (@SeanHayes, closes #1442)
  - Just a typo fix in the testing docs (by @bezidejni, closes #810)
  - Removed references to patterns() (by @SeanHayes, closes #1437)
  - Removed deprecated methods *Resource.apply_authorization_limits* and *Authorization.apply_limits* from code and documentation. (by @SeanHayes, closes #1383, #1045, #1284, #837)
  - Updates docs/cookbook.rst to make sure it's clear which *url* to import. (by @yuvadm, closes #716)
  - Updated docs/tutorial.rst. Without "null=True, blank=True" parameters in Slugfield, expecting "automatic slug generation" in save method is pointless. (by @orges, closes #753)
  - Cleaned up Riak docs. (by @SeanHayes, closes #275)

  - Include import statement for trailing_slash. (by @ljosa, closes #770)
  - Fix docs: *Meta.filtering* is actually a dict. (by @georgedorn, closes #807)
  - Fix load data command. (by @blite, closes #357, #358)
- Related schemas no longer raise error when not URL accessible. (Fixes PR #1439)
- Avoid modifying Field instances during request/response cycle. (closes #1415)
- Removing the Manager dependency in ToManyField.dehydrate(). (Closes #537)

v0.13.1

date 2016-01-25

Bugfixes

- Prevent muting non-tastypie's exceptions (#1297, PR #1404)
- Gracefully handle UnsupportFormat exception (#1154, PR #1417)
- Add related schema urls (#782, PR #1309)
- Repr value must be str in Py2 (#1421, PR #1422)
- Fixed assertHttpAccepted (PR #1416)

v0.13.0

date 2016-01-12

Dropped Django 1.5-1.6 support, added Django 1.9.

Bugfixes

- Various performance improvements (#1330, #1335, #1337, #1363)
- More descriptive error messages (#1201)
- Throttled requests now include Retry-After header. (#1204)
- In DecimalField.hydrate, catch decimal.InvalidOperation and raise ApiFieldError (#862)
- Add 'primary_key' Field To Schema (#1141)
- ContentTypes: Remove 'return' in __init__; remove redundant parentheses (#1090)
- Allow callable strings for ToOneField.attribute (#1193)
- Ensure Tastypie doesn't return extra data it received (#1169)
- In DecimalField.hydrate, catch decimal.InvalidOperation and raise ApiFieldError (#862)
- Fixed tastypie's losing received microseconds. (#1126)
- Data leakage fix (#1203)
- Ignore extra related data (#1336)

- Suppress Content-Type header on HTTP 204 (see #111) (#1054)
- Allow creation of related resources that have an 'items' related_name (supercedes #1000) (#1340)
- Serializers: remove unimplemented to_html/from_html (#1343)
- If GEOS is not installed then exclude geos related calls. (#1348)
- Fixed Resource.deserialize() to honor format parameter (#1354 #1356, #1358)
- Raise ValueError when trying to register a Resource class instead of a Resource instance. (#1361)
- Fix hydrating/saving of related resources. (#1363)
- Use Tastypie DateField for DateField on the model. (SHA: b248e7f)
- ApiFieldError on empty non-null field (#1208)
- Full schema (all schemas in a single request) (#1207)
- Added verbose_name to API schema. (#1370)
- Fixes Reverse One to One Relationships (Replaces #568) (#1378)
- Fixed "GIS importerror vs improperlyconfigured" (#1384)
- Fixed bug which occurs when detail_uri_name field has a default value (Issue #1323) (#1387)
- Fixed disabling cache using timeout=0, fixes #1213, #1212 (#1399)
- Removed Django 1.5-1.6 support, added 1.9 support. (#1400)
- stop using django.conf.urls.patterns (#1402)
- Fix for saving related items when resource_uri is provided but other unique data is not. (#1394) (#1410)

v0.12.2

date 2015-07-16

Dropped Python 2.6 support, added Django 1.8.

Bugfixes

- Dropped support for Python 2.6
- Added support for Django 1.8
- Fix stale data caused by prefetch_related cache (SHA: b78661d)

v0.12.1

date 2014-10-22

This release is a small bugfix release, specifically to remove accidentally added files in the Wheel release.

v0.12.0

date 2014-09-11

This release adds official support for both Django 1.7, as well as several bugfixes.

Warning: If you were previously relying on importing the `User` model from `tastypie.compat`, this import will no longer work correctly. This was removed due to the way app-loading works in Django 1.7 & no great solution for dodging this issue exists.

If you were using either of:

```
from tastypie.compat import User
from tastypie.compat import username_field
```

Please update your code as follows:

```
from tastypie.compat import get_user_model
from tastypie.compat import get_username_field
```

Bugfixes

- Drastic reworking of the test suite. (SHA: 95f57f7)
- Fixed Travis to run Python 3.4 tests. (SHA: 7af528c)
- Fixed a bug where URLs would be incorrectly handled if the `api_name` & `resource_name` were the same. (SHA: fd55aa3)
- Fixed a test requirement for PyYAML. (SHA: b4f6531)
- Added support for Django 1.7. (SHA: 7881bb6)
- Documentation updates:
  - Fixed header in `tools.rst`. (SHA: f8af772)
  - Fixed header in `resources.rst`. (SHA: 9508cbf)

v0.11.1

date 2014-05-22

This release is primarily a security release. The two issues fixed have been present but unknown for a long time & **ALL** users are recommended to upgrade where possible.

1. Tastypie previously would accept a relation URI & *solely* parse out the identifiers, ignoring if the URI was for the right resource. Where `'user': '/api/v1/users/1/',` would be accepted as a `User` URI, you could accidentally/intentionally pass something like `'user': '/api/v1/notes/1/',` (**notes** rather than **users**), which would assign it to the `User` with a `pk=1`. Tastypie would resolve the URI, but proceed to *only* care about the `kwargs`, not validating it was for the correct resource.

   Tastypie now checks to ensure the resolving resource has a matching URI, so these cases of mistaken identity can no longer happen (& with quicker lookups). Thanks to Sergey Orshanskiy for the report!

   Fixed in SHA: 6da76c6

2. In some browsers (specifically Firefox), it was possible to construct a URL that would include an XSS attack (specifically around the `offset/limit` pagination parameters). Firefox seems to evaluate the JSON returned, completing the attack. Safari & Chrome do not appear to be affected.

   Tastypie now escapes all error messages that could be returned to the user to prevent this kind of attack in the future. Thanks to Micah Hausler for the report!

   Fixed in SHA: ae515bd

Should you find a security issue in Tastypie, please report it to tastypie-security@googlegroups.com. Please **DO NOT** open GitHub issues or post the issues on the main Tastypie mailing list. Thanks!

Bugfixes

- Removed a mutable argument to `Serializer.serialize`. (SHA: fb7326d)
- Fixed the `unquote` import in `tastypie.contrib.gis`. (SHA: 1958df0)
- Enabled testing on Travis for Python 3.4. (SHA: 6596935)
- Documentation updates:
  - Fixed indentation in v0.9.16 release notes. (SHA: dd3725c)
  - Updated the v0.10.0 release notes. (SHA: e4c2455)
  - Fixed a Cookbook example to import `json` correctly. (SHA: bc7eb42)
  - Updated the non-ORM docs to match Resource. (SHA: da6a629)
  - Fixed grammar in Authorization docs. (SHA: 765ebf3)
  - Fixed a typo in Authorization docs. (SHA: 4818f08)
  - Updated the Cookbook to move an alternative approach to the correct place. (SHA: 803d679)
  - Updated the tutorial to import `slugify` from a more modern location. (SHA: 86bb5d9)
  - Fixed up inheritence in the Tools docs. (SHA: a1a2e64 & SHA: 12aa298)
  - Added a section about `httpie` to the Tools docs. (SHA: 5e49436)
  - Corrected the URL for `biplist`. (SHA: 859ce97)
  - Added Postman to the list of Tools. (SHA: b9f0dec)
  - Fixed a typo on the docs index. (SHA: 17e5a91)
  - Fixed incorrect apostrophes. (SHA: 635729c)
  - Fixed a typo in the Resources docs. (SHA: d789eea)

v0.11.0

date 2013-12-03

This release is a bugfix release. This also fixes installing Tastypie on Python 3 (due to the `python-mimeparse` dependency change).

This release is also the first to be released as a Python wheel as well. Ex:

```
pip install wheel
pip install --use-wheel django-tastypie
```

Bugfixes

- Now raises `ApiFieldError` when a datetime can't be parsed. (SHA: b59ac03)
- Reduced the length of the `ApiKey.key` field to 128. (SHA: 1cdf2c4)
- A bunch of test improvements. (SHA: 4320db4, SHA: f1bc584 & SHA: bd75a92)
- Updated `SimpleCache` to the post-Django 1.3 world. (SHA: 4963d97)
- Fixed a bug where `tastypie.utils.timezone.now` could throw an exception. (SHA: b78175d)
- Fixed the `python-mimeparse` dependency in `setup.py`. (SHA: 26dc473)
- Pinned to the beta of `python3-digest`. (SHA: cc8ef0f)
- Fixed `CacheThrottle` to work with Django's `DummyCache`. (SHA: 5b8a316)
- Improved the error message from `assertHttpAccepted`. (SHA: 7dbed92)
- Now raises `BadRequest` if we fail to decode JSON. (SHA: e9048fd)
- Removed a duplicate `if` check. (SHA: 823d007)
- Added further exception checking for new versions of `dateutil`. (SHA: 9739a35 & SHA: a8734cf)
- Removed `simplejson`. (SHA: ae58615)
- Removed old Django 1.4 code. (SHA: 3c7ce47)
- Documentation updates:
  - Added examples of how to implement throttling. (SHA: a305549)
  - Added docs on how to disable all list endpoints. (SHA: f4f4df2)
  - Added docs on GFKs. (SHA: 50189d7)
  - Added to the Tools docs. (SHA: bdde083 & SHA: 8c59e2c)
  - Added docs about running tests with `tox`. (SHA: f0222ea)
  - Fixed docs on resources. (SHA: d5a290b, SHA: 0c859bf, SHA: bc6defd & SHA: 95be355)
  - Added `defusedxml` to the docs. (SHA: 521a696)
  - Added to Who Uses. (SHA: dea9bf8)

v0.10.0

date 2013-08-03

This release adds official Python 3 support! This effort was partially funded by RevSys & the PSF, so much thanks to them!

Features

- Python 3 support! Not a lot to say, beyond there are a couple dependency changes. Please see the *Python 3 Support* docs!

v0.9.16

date 2013-08-03

This release is a bugfix release, the last one before Python 3 support will be added. This is the final release of Tastypie that is compatible with Django 1.4.

Features

- Added support for Unicode HTTP method names. (SHA: 2ebb362)
- Added a response for HTTP 422 Unprocessable Entity. (SHA: 7aadbf8)
- Added ISO-8601 strict support. (SHA: 3043140)

Bugfixes

- Added a `None` value for `username_field` in the compat module. (SHA: 569a72e)
- Fixed a bug where `obj_create` would check authorization twice. (SHA: 9a404fa)
- Fixed test case to no longer require a defined object class. (SHA: d8e250f)
- Fixed the signature on `dehydrate` in the GIS resource class. (SHA: f724919)
- Fixed a bug involving updating foreign relations on create. (SHA: 50a6741)
- Changed the PUT response code (with `always_return_data = True`) from 202 to 200. (SHA: abc0bef)
- Documentation updates:
  - Added an OAuth 2.0 example to the cookbook. (SHA: 7c93ae2)
  - Added a `detail_uri_name` to the Non-PK example in the cookbook. (SHA: 1fde565)
  - Added docs warning about `total_count` & performance overhead by default. (SHA: ebfbb7f)
  - Updated the Nested Resources example in the cookbook. (SHA: d582ead)
  - Added an example of converting a list of objects to JSON in the cookbook. (SHA: 2e81342)

v0.9.15

date 2013-05-02

This release is primarily a bugfix release. It makes using Tastypie with Django 1.5 more friendly, better exceptions

Features

- Drops Python 2.5 support. Yes, this is a feature that will pave the way for Tastypie on Python 3 support.
- Added `TASTYPIE_ABSTRACT_APIKEY`, which allows switching the `ApiKey` model out. (SHA: b8f4b9c)

Bugfixes

- Better support for Django 1.5:
  - Removed deprecation warnings (SHA: bb01761)
  - Numerous custom User improvements (SHA: d24b390)
- Expanded places `use_in` is used (SHA: b32c45)
- Authorization is now only called once with a full bundle (SHA: f06f41)
- Changed `for_list` to accept a boolean (SHA: 01e620)
- Only save related models that have changed (SHA: 6efdea)
- Better exception reporting, especially in conjunction with Sentry
  - (SHA: 3f9ce0)
  - (SHA: 4adf11)
- Configuration warning about `defusedxml` (SHA: aa8d9fd)
- Fixed a dependency in `setup.py` (SHA: cb0fe7)
- Release notes became a thing! Hooray! (SHA: 95e2499)
- Documentation updates:
  - Typo in `CleanedDataFormValidation` (SHA: bf252a)
  - CamelCase JSON (SHA: bf252a)
  - Docstring typo (SHA: e86dad)
  - Per-user resource (SHA: a77b28c)
  - Added more details about creating the `ApiKey` model (SHA: 80f9b8)

v0.9.14

date 2013-03-19

An emergency release **removing** a failed attempt at using `rose` to handle versioning.

Features

- None

Bugfixes

- Removed the dependency on `rose`, which wasn't effective at de-duplicating the version information. :(

v0.9.13

SECURITY HARDENING

The latest version of Tastypie includes a number of important security fixes and all users are strongly encouraged to upgrade.

Please note that the fixes might cause backwards incompatibility issues, so please check the upgrade notes carefully.

Security hardening improvements

- XML decoding has been wrapped in the defusedxml library
- XML requests may no longer include DTDs by default
- Deserialization will return HTTP 400 for any XML decode errors
- Don't even use XML and want to disable it? There's a simple *TASTYPIE_DEFAULT_FORMATS setting* setting to globally restrict the set of supported formats (closes #833):

  https://django-tastypie.readthedocs.io/en/v0.9.14/settings.html#tastypie-default-formats
- Content negotiation will return an error for malformed accept headers (closes #832)
- The Api class itself now allows a custom serializer (closes #817)
- The serialization documentation has been upgraded with security advice:

  https://django-tastypie.readthedocs.io/en/v0.9.14/serialization.html#serialization-security

Upgrade notes:

- **If you use XML serialization (enabled by default):**
  - defusedxml is now required
  - defusedxml requires lxml 3 or later

  ```
  pip install defusedxml "lxml>=3"
  ```
- Python 2.5 is no longer officially supported because defusedxml requires Python 2.6 or later. If you cannot upgrade to a newer version of Python please consider disabling XML support entirely.

CHAPTER 28

Quick Start

1. Add `tastypie` to `INSTALLED_APPS`.
2. Create an `api` directory in your app with a bare `__init__.py`.
3. Create an `<my_app>/api/resources.py` file and place the following in it:

```
from tastypie.resources import ModelResource
from my_app.models import MyModel


class MyModelResource(ModelResource):
    class Meta:
        queryset = MyModel.objects.all()
        allowed_methods = ['get']
```

4. In your root URLconf, add the following code (around where the admin code might be):

```
from django.conf.urls import url, include
from tastypie.api import Api
from my_app.api.resources import MyModelResource

v1_api = Api(api_name='v1')
v1_api.register(MyModelResource())

urlpatterns = [
  # ...more URLconf bits here...
  # Then add:
  url(r'^api/', include(v1_api.urls)),
]
```

5. Hit http://localhost:8000/api/v1/?format=json in your browser!

CHAPTER 29

Requirements

Core

- Python 2.7+ or Python 3.4+
- Django 1.8 or 1.11 (LTS releases)
- dateutil (http://labix.org/python-dateutil) >= 2.1

Format Support

- XML: lxml 3 (http://lxml.de/) and defusedxml (https://pypi.python.org/pypi/defusedxml)
- YAML: pyyaml (http://pyyaml.org/)
- binary plist: biplist (http://explorapp.com/biplist/)

Optional

- HTTP Digest authentication: python3-digest (https://bitbucket.org/akoha/python-digest/)

CHAPTER 30

Why Tastypie?

There are other API frameworks out there for Django. You need to assess the options available and decide for yourself. That said, here are some common reasons for tastypie.

- You need an API that is RESTful and uses HTTP well.
- You want to support deep relations.
- You DON'T want to have to write your own serializer to make the output right.
- You want an API framework that has little magic, very flexible and maps well to the problem domain.
- You want/need XML serialization that is treated equally to JSON (and YAML is there too).

CHAPTER 31

Reference Material

- https://django-tastypie.readthedocs.io/en/latest/
- https://github.com/django-tastypie/django-tastypie/tree/master/tests/basic shows basic usage of tastypie
- http://en.wikipedia.org/wiki/REST
- http://en.wikipedia.org/wiki/List_of_HTTP_status_codes
- http://www.ietf.org/rfc/rfc2616.txt
- http://jacobian.org/writing/rest-worst-practices/

CHAPTER 32

Getting Help

There are two primary ways of getting help.

1. Go to StackOverflow and post a question with the `tastypie` tag.
2. We have an IRC channel (#tastypie on irc.freenode.net) to get help, bounce an idea by us, or generally shoot the breeze.

CHAPTER 33

Running The Tests

The easiest way to get setup to run Tastypie's tests looks like:

```
$ git clone https://github.com/django-tastypie/django-tastypie.git
$ cd django-tastypie
$ virtualenv env
$ . env/bin/activate
$ ./env/bin/pip install -U -r requirements.txt
```

Then running the tests is as simple as:

```
# From the same directory as above:
$ ./env/bin/pip install -U -r tests/requirements.txt
$ ./env/bin/pip install tox
$ tox
```

Tastypie is maintained with all tests passing at all times for released dependencies. (At times tests may fail with development versions of Django. These will be noted as allowed failures in the `.travis.yml` file.) If you find a failure, please report it along with the versions of the installed software.

Python Module Index

t

Index

Symbols

A

B

C

T

U

W