Tastypie Documentation

Release 0.9.6

Daniel Lindsley, Cody Soyland & Matt Croydon

August 03, 2013

CONTENTS

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

GETTING STARTED WITH TASTYPIE

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

Not everyone's needs are the same, so Tastypie goes out of 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`:

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


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

    def __unicode__(self):
        return self.title

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

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

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

1.1 Installation

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

1. Download the dependencies:

- Python 2.4+

- Django 1.0+ (tested on Django 1.1+)

- `mimeparse` 0.1.3+ (http://code.google.com/p/mimeparse/)

    - Older versions will work, but their behavior on JSON/JSONP is a touch wonky.

- `dateutil` (http://labix.org/python-dateutil)

- **OPTIONAL** - `lxml` (http://codespeak.net/lxml/) if using the XML serializer

- **OPTIONAL** - `pyyaml` (http://pyyaml.org/) if using the YAML serializer

- **OPTIONAL** - `uuid` (present in 2.5+, downloadable from http://pypi.python.org/pypi/uuid/) if using the `ApiKey` authentication

2. 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.

1.2 Configuration

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

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

1.3 Creating Resources

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

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


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

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

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

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.

1.4 Hooking Up The Resource(s)

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

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

entry_resource = EntryResource()

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

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

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

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

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

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

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

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

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

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

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

1.5 Creating More Resources

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

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


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


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

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

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

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

1.6 Adding To The Api

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

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

```python
# urls.py
from django.conf.urls.defaults import *
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 = patterns('',
    # The normal jazz here...
    (r'^blog/', include('myapp.urls')),
    (r'^api/', include(v1_api.urls)),
)
```

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

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

- http://127.0.0.1:8000/api/v1/?format=json

- http://127.0.0.1:8000/api/v1/user/?format=json

- http://127.0.0.1:8000/api/v1/user/1/?format=json

- http://127.0.0.1:8000/api/v1/user/schema/?format=json

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

- http://127.0.0.1:8000/api/v1/entry/?format=json

- http://127.0.0.1:8000/api/v1/entry/1/?format=json

- http://127.0.0.1:8000/api/v1/entry/schema/?format=json

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

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

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

1.7 Limiting Data And Access

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

```python
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:

```python
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:

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

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

1.8 Beyond The Basics

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

- *Authentication / 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.

TASTYPIE SETTINGS

This is a comprehensive list of the settings Tastypie recognizes.

2.1 `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
```

Defaults to 20.

2.2 `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`.

2.3 `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."`.

2.4 `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.

> **Warning:** This setting causes the `Resource.get_multiple()` method to fail. If you need this method, you will have to override the URLconf to meet your needs.

An example:

```
TASTYPIE_ALLOW_MISSING_SLASH = True
```

Defaults to `False`.

2.5 `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` & `rfc-2822`.

An example:

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

Defaults to `iso-8601`.

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.

3.1 Quick Start

A sample resource definition might look something like:

```python
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'],
        }
```

3.2 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.

3.3 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, a search solution like Haystack or even managed filesystem data are all good use cases for `Resource` knowing nothing about the ORM.

3.4 Flow Through The Request/Response Cycle

TBD

3.5 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.

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

3.6 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.

3.7 Advanced Data Preparation

Tastypie uses a "dehydrate" cycle to prepare data for serialization & a "hydrate" cycle to take data sent to it & turn that back into useful Python objects.

Within these cycles, there are several points of customization if you need them.

3.7.1 `dehydrate`

3.7.2 `dehydrate_FOO`

3.7.3 `hydrate`

3.7.4 `hydrate_FOO`

3.8 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:

```python
# 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:

```python
# 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()
```

3.9 Resource Options (AKA `Meta`)

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

3.9.1 `serializer`

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

3.9.2 `authentication`

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

3.9.3 `authorization`

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

3.9.4 `validation`

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

3.9.5 `cache`

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

3.9.6 `throttle`

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

3.9.7 `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']` as a shortcut to prevent having to specify the other options.

3.9.8 `list_allowed_methods`

Controls what list REST methods the `Resource` should respond to. Default is `['get', 'post', 'put', 'delete']`.

3.9.9 `detail_allowed_methods`

Controls what list REST methods the `Resource` should respond to. Default is `['get', 'post', 'put', 'delete']`.

3.9.10 `limit`

Controls what 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.

3.9.11 `api_name`

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

3.9.12 `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`).

3.9.13 `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`.

3.9.14 `filtering`

Provides a list of 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.

3.9.15 `ordering`

Specifies the what fields the `Resource` should 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).

3.9.16 `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.

3.9.17 `queryset`

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

Unused by `Resource` but present for consistency.

3.9.18 `fields`

Controls what introspected fields the `Resource` should include. A whitelist of fields. Default is `[]`.

3.9.19 `excludes`

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

3.9.20 `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`.

3.9.21 `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`.

3.10 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:

```python
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
```

3.11 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 <http://haystacksearch.org>* 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:

```python
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
```

3.12 `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.

3.12.1 `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.

3.12.2 `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 (**NOT** wrapped in `patterns`).

3.12.3 `override_urls`

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

A hook for adding your own URLs or overriding 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 (**NOT** wrapped in `patterns`).

3.12.4 `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.

3.12.5 `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.

3.12.6 `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`.

3.12.7 `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`.

3.12.8 `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.

3.12.9 `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.

3.12.10 `dispatch`

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

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

3.12.11 `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))
```

3.12.12 `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'])
```

3.12.13 `is_authorized`

Resource.**is_authorized**(*self*, *request*, *object=None*)

Handles checking of permissions to see if the user has authorization to GET, POST, PUT, or DELETE this resource. If `object` is provided, the authorization backend can apply additional row-level permissions checking.

3.12.14 `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`.

3.12.15 `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`.

3.12.16 `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`.

3.12.17 `build_bundle`

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

3.12.18 `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`.

3.12.19 `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`.

3.12.20 `get_resource_uri`

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

This needs to be implemented at the user level.

A   `return reverse("api_dispatch_detail", kwargs={'resource_name':` `self.resource_name, 'pk':  object.id})` should be all that would be needed.

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

3.12.21 `get_resource_list_uri`

Resource.**get_resource_list_uri**(*self*)

Returns a URL specific to this resource's list endpoint.

3.12.22 `get_via_uri`

Resource.**get_via_uri**(*self*, *uri*)

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

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

3.12.23 `full_dehydrate`

Resource.**full_dehydrate**(*self*, *obj*)

Given an object instance, extract the information from it to populate the resource.

3.12.24 `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.

3.12.25 `full_hydrate`

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

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

3.12.26 `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.

3.12.27 `hydrate_m2m`

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

Populate the ManyToMany data on the instance.

3.12.28 `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.

3.12.29 `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.

3.12.30 `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.

3.12.31 `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`.

3.12.32 `apply_authorization_limits`

Resource.**apply_authorization_limits**(*self*, *request*, *object_list*)

Allows the `Authorization` class to further limit the object list. Also a hook to customize per `Resource`.

Calls `Authorization.apply_limits` if available.

3.12.33 `obj_get_list`

Resource.**obj_get_list**(*self*, *request=None*, ***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`.

3.12.34 `cached_obj_get_list`

Resource.**cached_obj_get_list**(*self*, *request=None*, ***kwargs*)

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

3.12.35 `obj_get`

Resource.**obj_get**(*self*, *request=None*, ***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`.

3.12.36 `cached_obj_get`

Resource.**cached_obj_get**(*self*, *request=None*, ***kwargs*)

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

3.12.37 `obj_create`

Resource.**obj_create**(*self*, *bundle*, *request=None*, ***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`.

3.12.38 `obj_update`

Resource.**obj_update**(*self*, *bundle*, *request=None*, ***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`.

3.12.39 `obj_delete_list`

Resource.**obj_delete_list**(*self*, *request=None*, ***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`.

3.12.40 `obj_delete`

Resource.**obj_delete**(*self*, *request=None*, ***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`.

3.12.41 `create_response`

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

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

Mostly a useful shortcut/hook.

3.12.42 `is_valid`

Resource.**is_valid**(*self*, *bundle*, *request=None*)

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.

3.12.43 `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`.

3.12.44 `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).

3.12.45 `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).

3.12.46 `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 `HttpAccepted` (204 No Content).

3.12.47 `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 an existing resource is modified, return `HttpAccepted` (204 No Content).

3.12.48 `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).

3.12.49 `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).

3.12.50 `delete_list`

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

Destroys a collection of resources/objects.

Calls `obj_delete_list`.

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

3.12.51 `delete_detail`

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

Destroys a single resource/object.

Calls `obj_delete`.

If the resource is deleted, return `HttpAccepted` (204 No Content). If the resource did not exist, return `HttpGone` (410 Gone).

3.12.52 `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).

3.12.53 `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).

3.13 `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.

3.13.1 `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.

3.13.2 `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.

3.13.3 `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.

3.13.4 `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.

3.13.5 `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.

3.13.6 `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 authorization or other overrides.

3.13.7 `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.

3.13.8 `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.

3.13.9 `obj_create`

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

A ORM-specific implementation of `obj_create`.

3.13.10 `obj_update`

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

A ORM-specific implementation of `obj_update`.

3.13.11 `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.

3.13.12 `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.

3.13.13 `rollback`

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

A ORM-specific implementation of `rollback`.

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

3.13.14 `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.

3.13.15 `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.

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.

4.1 Quick Start

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

```python
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 = patterns('',
    (r'^api/', include(v1_api.urls)),
)
```

4.2 `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.

4.2.1 `register`

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

Registers 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`.

4.2.2 `unregister`

`Api.unregister(self, resource_name):`

If present, unregisters a resource from the API.

4.2.3 `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):`

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

Should return a list of individual URLconf lines (**NOT** wrapped in `patterns`).

4.2.4 `urls`

`Api.urls(self):`

Property

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

4.2.5 `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.

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.

5.1 Usage

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

```python
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()
```

5.2 Caching Options

Tastypie ships with the following `Cache` classes:

5.2.1 `NoCache`

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

5.2.2 `SimpleCache`

This option does basic object caching, attempting to find the object in the cache & writing the object to the cache. It uses Django's current `CACHE_BACKEND` to store cached data.

5.3 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:

```python
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`.

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). This will be immediately returned to the user, serialized in the format they requested.

6.1 Usage

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

```python
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()
```

6.2 Validation Options

Tastypie ships with the following `Validation` classes:

6.2.1 `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`.

6.2.2 `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.

> **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:

```python
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)
```

6.3 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:

```python
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.

AUTHENTICATION / AUTHORIZATION

Authentication & authorization make up the components needed to verify that a certain user has access to the API and what they can do with it.

Authentication answers the question "can they see this data?" This usually involves requiring credentials, such as an API key or username/password.

Authorization answers the question "what objects can they modify?" This usually involves checking permissions, but is open to other implementations.

7.1 Usage

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

```python
from django.contrib.auth.models import User
from tastypie.authentication import BasicAuthentication
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.
        authentication = BasicAuthentication()
        authorization = DjangoAuthorization()
```

7.2 Authentication Options

Tastypie ships with the following `Authentication` classes:

7.2.1 `Authentication`

The no-op authentication option, the client is always allowed through. Very useful for development and read-only APIs.

7.2.2 `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.

7.2.3 `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`.

7.3 Authorization Options

Tastypie ships with the following `Authorization` classes:

7.3.1 `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.

7.3.2 `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.

7.3.3 `DjangoAuthorization`

The most advanced form of authorization, this checks the permission a user has granted to them (via `django.contrib.auth.models.Permission`). In conjunction with the admin, this is a very effective means of control.

7.4 Implementing Your Own Authentication/Authorization

Implementing your own `Authentication`/`Authorization` classes is a simple process. `Authentication` has two methods to override (one of which is optional but recommended to be customized) and `Authorization` has just one required method and one optional method:

```python
from tastypie.authentication import Authentication
from tastypie.authorization import Authorization


class SillyAuthentication(Authentication):
    def is_authenticated(self, request, **kwargs):
        if 'daniel' in request.user.username:
```

```python
            return True

        return False

    # Optional but recommended
    def get_identifier(self, request):
        return request.user.username


class SillyAuthorization(Authorization):
    def is_authorized(self, request, object=None):
        if request.user.date_joined.year == 2010:
            return True
        else:
            return False

    # Optional but useful for advanced limiting, such as per user.
    def apply_limits(self, request, object_list):
        if request and hasattr(request, 'user'):
            return object_list.filter(author__username=request.user.username)

        return object_list.none()
```

Under this scheme, only users with 'daniel' in their username will be allowed in, and only those who joined the site in 2010 will be allowed to affect data.

If the optional `apply_limits` method is included, each user that fits the above criteria will only be able to access their own records.

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.

The default `Serializer` supports the following formats:

- json
- jsonp
- xml
- yaml
- html

8.1 Usage

Using this class is simple. It is the default option on all `Resource` classes unless otherwise specified. The following code is a no-op, but demonstrate how you could use your own serializer:

```python
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()
```

8.2 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:

```python
import time
from tastypie.serializers import Serializer


class CustomJSONSerializer(Serializer):
    def to_json(self, data, options=None):
        options = options or {}

        # Add in the current time.
        data['requested_time'] = time.time()

        data = self.to_simple(data, options)
        return simplejson.dumps(data, cls=json.DjangoJSONEncoder, sort_keys=True)

    def from_json(self, content):
        data = simplejson.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:

```python
import csv
import StringIO
from tastypie.serializers import Serializer


class CSVSerializer(Serializer):
    formats = ['json', 'jsonp', 'xml', 'yaml', 'html', 'csv']
    content_types = {
        'json': 'application/json',
        'jsonp': 'text/javascript',
        'xml': 'application/xml',
        'yaml': 'text/yaml',
        'html': 'text/html',
        'csv': 'text/csv',
    }

    def to_csv(self, data, options=None):
        options = options or {}
        data = self.to_simple(data, options)
        raw_data = StringIO.StringIO()
        # Untested, so this might not work exactly right.
        for item in data:
            writer = csv.DictWriter(raw_data, item.keys(), extrasaction='ignore')
            writer.write(item)
        return raw_data

    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
```

8.3 `Serializer` Methods

A swappable class for serialization.

This handles most types of data as well as the following output formats:

```
* json
* jsonp
* xml
* yaml
* html
```

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.

8.3.1 `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.

8.3.2 `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".

8.3.3 `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".

8.3.4 `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".

8.3.5 `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.

8.3.6 `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.

8.3.7 `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).

8.3.8 `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.

8.3.9 `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".

8.3.10 `to_json`

`Serializer.to_json(self, data, options=None):`

Given some Python data, produces JSON output.

8.3.11 `from_json`

`Serializer.from_json(self, content):`

Given some JSON data, returns a Python dictionary of the decoded data.

8.3.12 `to_jsonp`

Serializer.to_jsonp(self, data, options=None):

Given some Python data, produces JSON output wrapped in the provided callback.

8.3.13 `to_xml`

Serializer.to_xml(self, data, options=None):

Given some Python data, produces XML output.

8.3.14 `from_xml`

Serializer.from_xml(self, content):

Given some XML data, returns a Python dictionary of the decoded data.

8.3.15 `to_yaml`

Serializer.to_yaml(self, data, options=None):

Given some Python data, produces YAML output.

8.3.16 `from_yaml`

Serializer.from_yaml(self, content):

Given some YAML data, returns a Python dictionary of the decoded data.

8.3.17 `to_html`

Serializer.to_html(self, data, options=None):

Reserved for future usage.

The desire is to provide HTML output of a resource, making an API available to a browser. This is on the TODO list but not currently implemented.

8.3.18 `from_html`

Serializer.from_html(self, content):

Reserved for future usage.

The desire is to handle form-based (maybe Javascript?) input, making an API available to a browser. This is on the TODO list but not currently implemented.

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.

9.1 Usage

To specify a throttle, add the `Throttle` class to the `Meta` class on the `Resource`:

```python
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)
```

9.2 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:

9.2.1 **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.

9.2.2 `CacheThrottle`

This uses just the cache to manage throttling. Fast but prone to cache misses and/or cache restarts.

9.2.3 `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.

9.3 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:

```python
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.

TASTYPIE COOKBOOK

10.1 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:

```python
class MyModelResource(Resource):
    class Meta:
        qs = MyModel.objects.all()

    def dehydrate(self, bundle):
        bundle.data['custom_field'] = "Whatever you want"
        return bundle
```

10.2 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:

```python
# views.py
from django.shortcuts import render_to_response
from myapp.api.resources import UserResource


def user_detail(request, username):
    ur = UserResource()
    user = ur.obj_get_detail(username=username)

    # Other things get prepped to go into the context then...

    return render_to_response('myapp/user_detail.html', {
        # Other things here.
        "user_json": ur.serialize(None, ur.full_dehydrate(obj=user), 'application/json'),
    })
```

10.3 Using Non-PK Data For Your URLs

By convention, `ModelResource``'s 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:

```python
# myapp/api/resources.py
class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()

    def override_urls(self):
        return [
            url(r"^(?P<resource_name>%s)/(?P<username>[\w\d_.-]+)/$" % self._meta.resource_name, sel
        ]
```

The added URLconf matches before the standard URLconf included by default & matches on the username provided in the URL.

You can also do "nested resources" (resources within another related resource) by lightly overriding the `dispatch` method in a similar way:

```python
# myapp/api/resources.py
class EntryResource(ModelResource):
    user = fields.ForeignKey(UserResource, 'user')

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

    def dispatch(self, request_type, request, **kwargs):
        username = kwargs.pop('username')
        kwargs['user'] = get_object_or_404(User, username=username)
        return super(EntryResource, self).dispatch(request_type, request, **kwargs)


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

entry_resource = EntryResource()

urlpatterns = patterns('',
    # The normal jazz here, then...
    (r'^api/(?P<username>\w+)/', include(entry_resource.urls)),
)
```

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.

11.1 "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: application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
```

Note that `application/xml` comes first, which is a format that Tastypie handles by default, hence why 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.

SITES USING TASTYPIE

The following sites are a partial list of people using Tastypie. I'm always interested in adding more sites, so please find me (`daniellindsley`) via IRC or start a mailing list thread.

12.1 LJWorld Marketplace

- http://www2.ljworld.com/marketplace/api/v1/?format=json

12.2 Forkinit

- http://forkinit.com/api/v1/?format=json

GETTING HELP

There are two primary ways of getting help. We have a mailing list hosted at Google (http://groups.google.com/group/django-tastypie/) and an IRC channel (#tastypie on irc.freenode.net) to get help, want to bounce idea or generally shoot the breeze.

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:

```python
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):

```python
from tastypie.api import Api
from my_app.api.resources import MyModelResource

v1_api = Api(api_name='v1')
v1_api.register(MyModelResource())

urlpatterns = patterns('',
  # ...more URLconf bits here...
  # Then add:
  (r'^api/', include(v1_api.urls)),
)
```

5. Hit http://localhost:8000/api/v1/?format=json in your browser!

REQUIREMENTS

Tastypie requires the following modules. If you use Pip, you can install the necessary bits via the included `requirements.txt`:

- Python 2.4+
- Django 1.0+
- mimeparse 0.1.3+ (http://code.google.com/p/mimeparse/)
    - Older versions will work, but their behavior on JSON/JSONP is a touch wonky.
- dateutil (http://labix.org/python-dateutil)
- lxml (http://codespeak.net/lxml/) if using the XML serializer
- pyyaml (http://pyyaml.org/) if using the YAML serializer

If you choose to use Python 2.4, be warned that you will also need to grab the following modules:

- uuid (present in 2.5+, downloadable from http://pypi.python.org/pypi/uuid/) if using the `ApiKey` authentication