TastyTopping Documentation

Release 1.2.5

Christian Boelsen

August 10, 2015

Contents

So, you've done all that hard work creating your REST API using Tastypie on the server-side - why would you want to have to do it all again on the client-side?

TastyTopping creates objects that behave similarly to django models, using your self-documenting TastyPie REST API to create the object's attributes and behaviour, in addition to retrieving the data. TastyTopping only needs 2 things from you: The URL of the API, and the resource name.

As a brief example:

```
>>> factory = ResourceFactory('http://localhost/app_name/api/v1')
>>> ex = factory.example(field1='name', field2=10)
>>> ex.field3 = datetime.now()
>>> print ex.field4
1.234
```

CHAPTER 1

Contents

1.1 Getting Started

TastyTopping works together with a Tastypie API to easily access data remotely over HTTP. What makes TastyTopping so useful is its simplicity.

This tutorial is designed to work with the simple blog application found in Tastypie's tutorial. It also assumes you have some knowledge of Tastypie, so if you're not clear on Django, Tastypie, etc., then you'll probably want to look there first.

1.1.1 Installation

Installation is the usual simple affair with python nowadays:

1. Download the dependencies:
   - Python 2.7+ or Python 3.3+
   - requests 1.2.3+
2. Either check out TastyTopping from github or pull a release off PyPI: `pip install tastytopping`.

1.1.2 Usage

The *`tastytopping.ResourceFactory`* class is how we will access our API's resources. To begin with, it needs the URL of our API, which it takes in its constructor. After that, resources are accessed as members of the factory. Following Tastypie's simple blog application tutorial, let's add an entry to the blog:

```
from tastytopping import ResourceFactory
factory = ResourceFactory('http://127.0.0.1:8000/api/v1/')
# Get the first user (we don't mind which it is).
existing_user = factory.user.all().first()
new_entry = factory.entry(
    user=existing_user,
    title='New blog entry',
    body='Some text for the blog entry.\n'
)
new_entry.save()
```

To edit the blog entry at a later date, we simply need to edit the body field:

```
new_entry.body += 'EDIT: Some more text!\n'
new_entry.save()
```

Be aware that like the get() method on Django models, *get()* expects a single result to be returned, and will raise an exception otherwise (see *NoResourcesExist* and *MultipleResourcesReturned*).

Now that we've made the new blog entry, you'll probably notice it's not a very good blog entry - let's get rid of it:

```
new_entry.delete()
```

1.1.3 Beyond The Basics

That's all there is to the basics of TastyTopping. It's simple, and hopefully you'll find it useful.

That said, there is more to learn, if you need to use more of Tastypie's features:

- QuerySets
- Authentication
- Nested Resources
- Optimization
- TastyTopping Cookbook

1.2 Authentication

TastyTopping supports Tastypie's authentication types, and can be set per API and per Resource. The auth classes all inherit from requests's AuthBase, which means you can also use their excellent documentation.

For information on how authentication works in Tastypie, see their docs.

1.2.1 Usage

To use an authentication class is as simple as setting the 'auth' member on a *ResourceFactory* or a *Resource*. As an example, to use API key authentication for all Resources by default:

```
from tastytopping import ResourceFactory, HTTPApiKeyAuth
factory = ResourceFactory('http://localhost:8000/myapp/api/v1/')
factory.auth = HTTPApiKeyAuth(username, api_key)
```

And to use digest authentication on a single resource (secret_resource):

```
from tastytopping import HTTPDigestAuth
factory.secret_resource.auth = HTTPDigestAuth(username, password)
```

There is also a class to use with django's session authentication. This requires that you set up a Resource in tastypie that is capable of returning a CSRF token in a cookie, an example of which can be found in the django app used by TastyTopping's unit tests.

Once a CSRF token has been returned in a cookie, telling a Resource to use session auth is as simple as:

```
from tastytopping import HTTPSessionAuth
factory.some_resource.auth = HTTPSessionAuth()
```

The CSRF token will be taken from the cookies automatically. If the CSRF token was obtained in another way, it's also possible to pass the token into *HTTPSessionAuth*'s constructor.

Besides the aforementioned auth classes, TastyTopping also provides *HTTPBasicAuth*. To use OAuth with your API, the requests-oathlib package provides a compatible authentication class.

1.2.2 Do It Yourself

If it turns out that you need to implemente your own authentication class on the server-side, or you're simply using one that isn't included in TastyTopping, then it's always possible to roll your own authentication class.

For documentation on how to do just that, see the excellent docs provided by requests on the subject. For an example of how to make an authentication class that interacts with a Tastypie Resource, see the HTTPApiKeyAuth class on github.

1.3 QuerySets

QuerySets are a way to contruct queries for a particular resource, while minimising requests sent to the API. They function similarly to Django's QuerySets, so if you're comfortable with these you should be right at home with TastyTopping's QuerySets (differences will be highlighted as you go).

Like Django's QuerySets, these will be evaluated only when needed - a term that has a slightly different meaning here - in the following cases:

- **Iteration.** A QuerySet is iterable, and it executes its database query the first time you iterate over it.
- **Slicing / Indexing.** Unlike with Django's, TastyTopping's QuerySets are always evaluated when slicing or indexing.
- **list().** Same warnings apply as with Django's QuerySets - using `list()` will iterate over the whole QuerySet and load it into memory.
- **bool().** Testing a QuerySet in a boolean context, such as using `bool()`, `or`, `and` or an `if` statement, will cause the query to be executed.

1.3.1 Creation

To create a `QuerySet`, it's usually easiest to use one of the methods from *Resource* which returns a QuerySet for that Resource:

- *all()* - return a QuerySet matching all objects of a particular Resource.
- *filter()* - return a QuerySet matching objects filtered by the given keyword args. The filters used are the same as those passed to tastypie.
- *none()* - return an EmptyQuerySet. It contains shortcuts to avoid hitting the API where necessary.

1.3.2 Usage

To demonstrate using QuerySets, we're going to use the same API as in the Getting Started section:

```
from tastytopping import ResourceFactory
factory = ResourceFactory('http://127.0.0.1:8000/api/v1/')
```

We've even already used a QuerySet as part of the tutorial:

```
existing_user = factory.user.all().first()
```

which is simple enough - it will get the first user (using the default ordering). Using this existing user, we can query the API to see how many blog entries this user has made:

```
blog_entries = factory.entry.filter(user=existing_user)
num_blog_entries = blog_entries.count()
```

To update the published date on all of these blog entries to the current date in a single call:

```
from datetime import datetime
blog_entries.update(pub_date=datetime.now())
```

To delete all blog entries from before 2012:

```
factory.entry.filter(pub_date__lt=datetime(2012)).delete()
```

There's a more convenient way to order the resources too; to order the blog entries by reverse date:

```
factory.entry.all().order_by('-pub_date')
```

or to get the latest blog entry:

```
factory.entry.all().latest('pub_date')
```

There are some optimizations possible with QuerySets:

- *Prefetching a QuerySet's related resources*.

To view all available methods, take a look at the API Documentation.

1.4 Nested Resources

Nested resources allow you to extend the functionality of a tastypie Resource in a nice and simple way. It would make sense to access that nested resource on the client-side in a nice and simple way too, which is exactly what TastyTopping does. For information on how to create nested resources in tastypie, check out tastypie's docs and TastyTopping's unit test webapp.

1.4.1 Usage

A Resource's nested resources are accessible via the 'nested' attribute. Any attribute that accessed from 'nested' will be assumed to be a nested resource, since there's no standard way of accessing that information via a schema.

The examples below will illustrate what's configured on the server side by showing the contents of a Resource's `prepend_urls()` method. Nested resources can be appended to both the list view and detail view, so we'll go through a couple of examples of each.

List View

```
# api.py on the server-side
def prepend_urls(self):
    return [
        url(
            r'^(?P<resource_name>{0})/add/(?P<num1>\d+)/(?P<num2>\d+){1}$'.format(self._meta.resource
            self.wrap_view('calc_add'),
```

```
            name="api_calc_add",
        ),
    ]
```

So, in this (silly) example, we've got a nested resource at `<resource_name>/add/<num1>/<num2>/`, which we'll access with a GET:

```
# client-side
factory = ResourceFactory('http://some-server.com/api/v1/')
my_sum = factory.some_resource.nested.add(2, 3).get()
```

This will send a GET request to `/api/v1/some_resource/add/2/3/`. A NestedResource will accept any `*args` and just append them to the URL. So:

```
factory.some_resource.nested.add(2, 3, 4, 5).get()
```

will send a GET request to `/api/v1/some_resource/add/2/3/4/5/`. In this case there's no matching URL on the server side, and you'll get an exception.

It's also possible to send POST, PUT, PATCH, and DELETE requests. In these instances it makes more sense to send any extra information as data:

```
# api.py on the server-side
def prepend_urls(self):
    return [
        url(
            r'^(?P<resource_name>{0})/mult{1}$'.format(self._meta.resource_name, trailing_slash()),
            self.wrap_view('calc_mult'),
            name="api_calc_mult",
        ),
    ]
```

What you can't see here is that `calc_mult()` accepts only POST requests, and expects two numbers as part of a dict (for example `{num1: 1, num2: 2}`) in the request's data. With that in mind, sending a request to this nested resource using TastyTopping looks like:

```
# client-side
factory = ResourceFactory('http://some-server.com/api/v1/')
my_product = factory.some_resource.nested.mult(num1=2, num2=3).post()
```

This will send a POST request to `/api/v1/some_resource/mult/`, and include the kwargs as the data dictionary.

Detail View

Now we'll take a look at a nested resource as part of a Resource's detail view. On tastypie's side, the matching regex will include the `pk`, which will be passed to the called method:

```
# api.py on the server-side
def prepend_urls(self):
    return [
        url(
            r'^(?P<resource_name>{0})/(?P<pk>\w[\w/-]*)/chained/getting/child{1}$'.format(self._meta
            self.wrap_view('get_child'),
            name="api_get_child",
        ),
    ]
```

`get_child()` expects no arguments, and will return a resource. To send a GET request to this nested resource is as simple as:

```
# client-side
factory = ResourceFactory('http://some-server.com/api/v1/')
a_resource = factory.some_resource.all().first()
child = a_resource.nested.chained.getting.child.get()
# Or, if you find it more readable:
child = a_resource.nested.chained.getting('child').get()
```

1.5 Optimization

Because TastyTopping communicates over the network to a tastypie API, operations are expensive. Care has been taken to only prod the API when needed, but when dealing with thousands of resources a bit of extra help would be nice. Thankfully, there are a few ways to optimize these network accesses.

1.5.1 Bulk operations (PATCH)

The PATCH REST method provides a nice way to for clients to create, update and delete resources en masse, which tastypie has implemented. TastyTopping has wrapped this functionality behind the *bulk()* method, with convenience methods provided for readability and ease of use (*create()*, *update()*, *delete()*,). To create multiple blogentries (from the tutorial):

```
factory.entry.create([
    {user=user1, title='Entry 1', body='Some text.\n'},
    {user=user1, title='Entry 2', body='More text.\n'},
    {user=user2, title='Entry 3', body='This text.\n'},
    {user=user1, title='Entry 4', body='... text.\n'},
])
```

Note that unlike when creating a resource normally, the *create()* method does NOT return anything. This means if you want to get any of the resources later, you'll need to *get()* them.

Using a *QuerySet*, it's also possible to update multiple Resources in just two requests:

```
queryset = factory.entry.filter(title__in=['Entry1', 'Entry2'])
queryset.update(user=user2)
```

Note that while the update only takes a single request, there is a previous request that will GET the relevant objects to update (seeing as it's not possible to do it in one request like with SQL), because we need to know the URIs of all relevant resources.

Lastly, it's possible to delete multiple Resources in two requests (GET and PATCH like with *update()*):

```
# From the previous example.
queryset.delete()
```

The exception to `delete()` requiring two requests is when the QuerySet contains all resources (ie. you used *all()*). Then a DELETE is sent to the requests list view.

If you really needed to remove every last possible request, you can also combine all the previous calls into a single bulk() call:

```
entry3.user = user1
factory.entry.bulk(create=[
        {user=user1, title='Entry 5', body='Some text.\n'},
```

```
        {user=user1, title='Entry 6', body='More text.\n'},
    ],
    update=[entry3],
    delete=[entry4]
)
```

You might now be thinking that this sounds pretty good. You might even be thinking that you'll use this wherever possible. Well, there is a single, potentially bad, downside: Because of the potentially large size of bulk updates, the API will respond with a 202 before completing the request (see wikipedia, and tastypie). This means it's possible for the request to fail without us knowing. However, in the event that it does fail, all changes will be rolled back.

1.5.2 Update multiple fields

As a shortcut, it's possible to update multiple fields in a single request using *update()*, which will also update the resource remotely (ie. effectively call *save()*).

```
entry1.update(
    user=user2,
    title='Different title',
    body='Different text',
)
```

1.5.3 Prefetching a QuerySet's related resources

For a *QuerySet* that returns a large number of resources, it is sometimes more efficient to prefetch some, or all, of the resources' related resources. This can be achieved using a QuerySet's *prefetch_related()* method, which will GET all resources of the given type in a single request and perform an SQL-type 'join'.

Take the example below, which will loop through all collections (a made-up resource that contains many blog entries) and print the title of each blog entry in the collection:

```
collection_queryset = factory.collection.all()
for collection in collection_queryset:
    for entry in collection.entries:
        print(entry.title)
```

In this case, there will be an initial GET request for the collections, followed by a GET request for each `entry` in the collection. Ouch!

To get around this situation, you can call *prefetch_related()* on the initial QuerySet:

```
collection_queryset = factory.collection.all()
collection_queryset.prefetch_related('entries')
for collection in collection_queryset:
    for entry in collection.entries:
        print(entry.title)
```

This time, there will be a grand total of two GET requests: one for the collections, and one for the entries.

There is a trade-off with this method, though, and that is that every resource of the requested type will be prefetched. This means that if you only need to prefetch a few resources, or there are a lot of resources of the requested type, then it can also be detrimental to call *prefetch_related()*.

1.5.4 Server-side

There are several ways to reduce the number of requests sent to the API by setting up your tastypie Resources (possibly) differently. As always, don't use these suggestions where they don't make sense (ie. use your brain!).

max_limit

In a tastypie Resource, there is a member of the `Meta` class called `max_limit`. Because TastyTopping only fetches the resources which were queried, it's advisable to set this to `0`, to minimise requests sent (or at least sufficiently large, if not unlimited). The `limit` member should still be set to a reasonably small value (like the default of `20`), since that is used when iterating over a QuerySet.

always_return_data

Setting `always_return_data = True` will ensure a resource's details are returned from a POST request when creating it. If this is set to `False`, TastyTopping needs to transmit another GET request when a Resource's fields are accessed.

1.6 TastyTopping Cookbook

1.6.1 Extending Resources

Since the *ResourceFactory* returns classes for a resource's list view, it's possible to inherit from these to extend their functionality. For instance, if you want each Resource to keep track of their lifetime on the client:

```
factory = ResourceFactory('http://localhost:8000/api/v1/')

class SomeResource(factory.some_resource):

    def __init__(self, *args, **kwargs):
        super(SomeResource, self).__init__(*args, **kwargs)
        # This is a field in the DB model:
        self.alive = True

    def __del__(self):
        self.alive = False
        self.save()
```

And then you can use the derived class as you would any class returned from the *ResourceFactory*:

```
new_resource = SomeResource(field1='value1', field2=2).save()
```

1.6.2 Ignore MultipleResourcesReturned when creating a Resource

So, you've arrived here after getting a *MultipleResourcesReturned* exception when trying to create a new Resource (or maybe you're just reading through the docs)? This section goes through what happens when creating a Resource without a unique field set, and what you can do about it.

Take a Resource whose only unique field is the auto-incrementing `id` field. Assuming the Resource has `always_return_data = False`, then creating two resources as below will create some problems:

```
factory.another_resource(name='Bob').save()
factory.another_resource(name='Bob').save()
```

The second *save()* will raise a *MultipleResourcesReturned* exception. This happens because TastyTopping will attempt to GET the newly created resource. The response, however, will return two resources, which means TastyTopping can't be sure which one it created.

As suggested by the exception, one easy way around this, especially if you don't use the Resources after creating them, is to use :*create()*:

```
factory.another_resource.create([
    {'name': 'Bob'},
    {'name': 'Bob'},
])
```

No attempt will be made to GET the resources after POSTing them, so the problem won't be encountered.

The other solution presented by the exception is to explicitly ignore the exception and GET the latest resource anyway:

```
factory.another_resource(name='Bob').save()
try:
    new_resource = factory.another_resource(name='Bob').save()
except MultipleResourcesReturned:
    new_resource = factory.another_resource.filter(name='Bob').latest()
```

Be warned, though, that this should only be done if you are SURE that no other Resource was created in the meantime, either in another thread, another process, or another machine.

1.7 API Documentation

1.7.1 ResourceFactory

class tastytopping.**ResourceFactory**(*api_url*, *verify=True*)

Create classes with which to access the API's resources.

The resource classes are accessed as member variables on the factory object, via a resource's name. For example, with a resource at `http://localhost/app_name/api/v1/example_resource/`, the ResourceFactory will have a member variable called 'example_resource' returning a *Resource* class (more specifically, a subclass of it, specialised for the resource in question):

```
>>> factory = ResourceFactory('http://localhost/app_name/api/v1/')
>>> old_resource = factory.example_resource.get(name='bob')
>>> new_resource = factory.example_resource(name='new name')
>>> # And to see what resources are available:
>>> factory.resources
['example', 'another_resource', 'entry']
```

Parameters

- **api_url** (*str*) – The url of the API!
- **verify** (*bool*) – Sets whether SSL certificates for the API should be verified.

Variables **resources** – (list) - The names of each *Resource* this factory can create.

add_factory_dependency(*factory*)

Add another ResourceFactory as a dependency.

If any of the Resources associated with this ResourceFactory (Api on the tastypie side) have foreign keys to Resources associated with a different ResourceFactory, then this ResourceFactory depends on the other to create the Resources. In this case, you will need to pass the other ResourceFactory into this method.

Parameters **factory** (ResourceFactory) – The ResourceFactory that this depends on.

auth
(*AuthBase*) - Update the auth on all resources accessed via this API. Any new Resources will have their auth set to this value too.

1.7.2 Resource

class tastytopping.resource.**Resource**(***kwargs*)
A base class to inherit from, to wrap a TastyPie resource.

To wrap a TastyPie resource, a class must be defined that inherits from Resource. This derived class must specify, as a minimum, the class members 'api_url', and 'resource_name' (see below for descriptions). *ResourceFactory* returns instances of this class from its methods. Users are strongly encouraged to use these factory methods instead of directly subclassing from Resource.

Parameters **kwargs** (*dict*) – Keyword arguments detailing the fields for the new resource.

classmethod all()
Returns a QuerySet with no filters applied.

Returns A new QuerySet.

Return type *QuerySet*

classmethod bulk(*create=None*, *update=None*, *delete=None*)
Create, update, and delete to multiple resources in a single request.

Note that this doesn't return anything, so any created resources will have to be retrieved with *get()* / *update()* / *all()*. Resource objects passed into delete will be marked as deleted, so any attempt to use them afterwards will raise an exception.

Because of the potentially large size of bulk updates, the API will respond with a 202 before completing the request (see wikipedia, and tastypie). This means it's possible for the request to fail without us knowing. So, while this method can be used for a sizeable optimization, there is a pitfall: You have been warned!

Parameters

- **create** (*list*) – The dicts of fields for new resources.
- **update** (*list*) – The Resource objects to update.
- **delete** (*list*) – The Resource objects to delete.

Raises *ResourceDeleted*

check_alive()
Check that the Resource has not been deleted.

Note that this only checks locally, so if another client deletes the resource originating from another ResourceFactory, or a different PC, it won't be picked up.

Raises *ResourceDeleted*

classmethod create(*resources*)
Creates new resources for each dict given.

This method exists purely for convenience and readability - internally it uses *bulk()*.

Parameters **resources** (*list*) – A list of fields (dict) for new resources.

delete()
Delete the object through the API.

Note that any attempt to use this object after calling delete will result in an ResourceDeleted exception.

Raises *ResourceDeleted*

fields()
Return the fields according to the API.

Returns The resource's fields as {name (str): value (object)}.

Return type dict

classmethod filter(***kwargs*)
Return a QuerySet, with the given filters applied.

Parameters **kwargs** (*dict*) – Keywors arguments to filter the search.

Returns A new QuerySet.

Return type *QuerySet*

classmethod get(***kwargs*)
Return an existing object via the API.

Parameters **kwargs** (*dict*) – Keywors arguments to filter the search.

Returns The resource identified by the kwargs.

Return type *Resource*

Raises *NoResourcesExist*, *MultipleResourcesReturned*

classmethod none()
Return an EmptyQuerySet object.

Returns A new QuerySet.

Return type *QuerySet*

refresh()
Retrieve the latest values from the API with the next member access.

save()
Saves a resource back to the API.

Raises *ResourceDeleted*

update(***kwargs*)
Set multiple fields' values at once, and call *save()*.

Parameters **kwargs** (*dict*) – The fields to update as keyword arguments.

uri()
Return the resource_uri for this object.

Returns resource_uri

Return type str

Raises *ResourceHasNoUri*

1.7.3 QuerySet

class tastytopping.queryset.**QuerySet**(*resource*, ***kwargs*)

Allows for easier querying of resources while reducing API access.

The API and function are very similar to Django's QuerySet class. There are a few differences: slicing this QuerySet will always evaluate the query and return a list; and this QuerySet accepts negative slices/indices [1].

Note that you normally wouldn't instantiate QuerySets yourself; you'd be using a Resource's *filter()*, *all()*, *none()*, *get()* methods to create a QuerySet.

A quick example:

```
# These will not evaluate the query (ie. hit the API):
some_resources_50_100 = SomeResource.filter(rating__gt=50, rating__lt=100)
some_resources_ordered = some_resources_50_100.order_by('rating')

# These will evaluate the query:
first_resource_above_50 = some_resources_ordered.first()
arbitrary_resource_between_50_and_100 = some_resources_ordered[5]
all_resources_between_50_and_100 = list(some_resources_ordered)
every_third_resource_between_100_and_50 = some_resources_ordered[::-3]
```

all()

Returns a copy of this QuerySet.

Returns A new QuerySet.

Return type *QuerySet*

count()

Return the number of records for this resource.

Parameters **kwargs** (*dict*) – Keywors arguments to filter the search.

Returns The number of records for this resource.

Return type int

delete()

Delete every Resource filtered by this query.

Note that there is an optimization when calling delete() on a full QuerySet (ie. one without filters). So:

```
# this will be quicker:
Resource.all().filter()
# than this:
Resource.filter(id__gt=0).filter()
```

earliest(*field_name*)

Works otherwise like *latest()* except the direction is changed.

exists()

Returns whether this query matches any resources.

Returns True if any resources match, otherwise False.

Return type bool

[1] Using negative slices/indices will result in more requests to the API, as the QuerySet needs to find the number of resources this query matches (using *count()*).

filter(***kwargs*)
Return a new QuerySet, with the given filters additionally applied.

Parameters **kwargs** (*dict*) – Keywors arguments to filter the search.

Returns A new QuerySet.

Return type *QuerySet*

first()
Return the first resource from the query.

Returns The first Resource, or None if the QuerySet is empty.

Return type *Resource*

get(***kwargs*)
Return an existing object via the API.

Parameters **kwargs** (*dict*) – Keywors arguments to filter the search.

Returns The resource identified by the kwargs.

Return type *Resource*

Raises *NoResourcesExist*, *MultipleResourcesReturned*

iterator()
Returns an iterator to the QuerySet's results.

Evaluates the QuerySet (by performing the query) and returns an iterator over the results. A QuerySet typically caches its results internally so that repeated evaluations do not result in additional queries. In contrast, iterator() will read results directly, without doing any caching at the QuerySet level (internally, the default iterator calls iterator() and caches the return value). For a QuerySet which returns a large number of objects that you only need to access once, this can result in better performance and a significant reduction in memory.

Note that using iterator() on a QuerySet which has already been evaluated will force it to evaluate again, repeating the query.

Returns An iterator to the QuerySet's results.

Return type iterator object

last()
Works like *first()*, but returns the last resource.

latest(*field_name*)
Returns the latest resource, by date, using the 'field_name' provided as the date field.

Note that *earliest()* and *latest()* exist purely for convenience and readability.

Parameters **field_name** (*str*) – The name of the field to order the resources by.

Returns The latest resource, by date.

Return type *Resource*

Raises *NoResourcesExist*

none()
Return an EmptyQuerySet object.

order_by(**args*)
Order the query's result according to the fields given.

The first field's order will be most important, with the importance decending thereafter. Calling this method multiple times will achieve the same. For example, the following are equivalent:

```
query = query.order_by('path', 'content')
# Is equivalent to:
query = query.order_by('path')
query = query.order_by('content')
```

Parameters **args** (*tuple*) – The fields according to which to order the Resources.

Returns A new QuerySet.

Return type *QuerySet*

prefetch_related(**args*)

Returns a QuerySet that will automatically retrieve, in a single batch, related objects for each of the specified lookups.

This method simulates an SQL 'join' and including the fields of the related object, except that it does a separate lookup for each relationship and does the 'joining' in Python.

It will check that the related field hasn't already been 'joined' by setting 'full=True' in the Resource's field in tastypie.

Take note that this method will fetch all the resources of all the given fields to do the 'joining', so it only makes sense for QuerySets that will return a large nunmber of resources. Even then, watch the memory usage!

Parameters **args** (*tuple*) – The fields to prefetch.

Returns A new QuerySet.

Return type *QuerySet*

reverse()

Reverse the order of the Resources returned from the QuerySet.

Calling reverse() on an alerady-reversed QuerySet restores the original order of Resources.

Evaluating a QuerySet that is reversed but has no order will result in a *OrderByRequiredForReverse* exception being raised. So, ensure you call *order_by()* on any reversed QuerySet.

Returns A new QuerySet.

Return type *QuerySet*

update(***kwargs*)

Updates all resources matching this query with the given fields.

This method provides a large optimization to updating each resource individually: This method will only make 2 API calls per thousand resources.

Parameters **kwargs** (*dict*) – The fields to update: {field_name: field_value, ...}

1.7.4 Authentications

class tastytopping.auth.**AuthBase**

Base class that all auth implementations derive from

class tastytopping.auth.**HTTPApiKeyAuth**(*username*, *key*)

Use TastyPie's ApiKey authentication when communicating with the API.

class tastytopping.auth.**HTTPSessionAuth**(*csrf_token=None*)
Use Django's Session authentication when communicating with the API.

The CSRF token can either be passed in on construction, or it will be automatically taken from the session's cookies. If no CSRF token can be found, a *MissingCsrfTokenInCookies* exception will be raised.

extract_csrf_token(*cookies*)
Get the CSRF token given a session's cookies.

Parameters **cookies** (CookieJar) – A session's cookies, one of which should contain the CSRF token.

Raises *MissingCsrfTokenInCookies*

class tastytopping.auth.**HTTPBasicAuth**(*username*, *password*)
Attaches HTTP Basic Authentication to the given Request object.

class tastytopping.auth.**HTTPDigestAuth**(*username*, *password*)
Attaches HTTP Digest Authentication to the given Request object.

build_digest_header(*method*, *url*)

handle_401(*r*, ***kwargs*)
Takes the given response and tries digest-auth, if needed.

handle_redirect(*r*, ***kwargs*)
Reset num_401_calls counter on redirects.

1.7.5 Exceptions

exception tastytopping.exceptions.**BadUri**
Raised when the URI given does not belong to the API.

exception tastytopping.exceptions.**CannotConnectToAddress**
Raised when no connection was possible at the given address.

exception tastytopping.exceptions.**CreatedResourceNotFound**
Raised when no resource can be found matching the resource created.

exception tastytopping.exceptions.**ErrorResponse**
Raised when an error status is returned from the API.

exception tastytopping.exceptions.**FieldNotInSchema**
Raised when a field should be part of the resource's schema, but isn't.

exception tastytopping.exceptions.**FieldNotNullable**
Raised when attempting to set a field to None, when the API forbids it.

exception tastytopping.exceptions.**FilterNotAllowedForField**
Raised when the filter used is not in the list of filters for the field in the API.

exception tastytopping.exceptions.**IncorrectNestedResourceArgs**
Raised when failing to GET a nested resource.

This is caused by tastypie raising a NotFound error in a 202 response. The cause is (almost always) an incorrect number of args to the method.

exception tastytopping.exceptions.**IncorrectNestedResourceKwargs**
Raised when failing to GET a nested resource.

Specifically, a MultiValueDictKeyError was raised in the nested resource. Since kwargs should have been passed to the Resource method, which the nested resource should be retrieving from the request.GET dict, it is assumed that kwargs were missing.

exception tastytopping.exceptions.**InvalidFieldName**
Raised when a field name will cause unexpected behaviour.

For instance, if a field is called 'limit', or 'order_by', it won't be possible to order or limit the search results.

exception tastytopping.exceptions.**InvalidFieldValue**
Raised when a field has been passed the wrong type.

exception tastytopping.exceptions.**MissingCsrfTokenInCookies**
Raised when no CSRF token could be found in a session's cookies.

This exception normally occurs when no CSRF token was passed to a HTTPSessionAuth object and there was no user authentication prior (which returned a CSRF token).

exception tastytopping.exceptions.**MultipleResourcesReturned**
Raised when more than one resource was found where only one was expected.

exception tastytopping.exceptions.**NoDefaultValueInSchema**
Raised when a field has no default value, but the user asked for one.

Note that this can happen if you haven't yet saved a Resource, and you're using a field that you haven't provided a value for. For instance:

```
>>> res = factory.test_resource(path='test/path')
>>> res.rating      # Exception raised if rating has no default value.
```

exception tastytopping.exceptions.**NoFiltersInSchema**
Raised when the resource has no filters listed in the schema.

exception tastytopping.exceptions.**NoResourcesExist**
Raised when getting resources, but none were found.

exception tastytopping.exceptions.**NoUniqueFilterableFields**
Raised when the object has no fields with unique values to filter on.

exception tastytopping.exceptions.**OrderByRequiredForReverse**
Raised by QuerySet when attempting to reverse a query without an order.

This exception will be raised when attempting to evaluate a QuerySet that should be reversed (ie. reverse() has been called at least once), but does not have an order.

exception tastytopping.exceptions.**PrettyException**
Ensure the JSON dicts fed into the exceptions are formatted nicely.

exception tastytopping.exceptions.**ReadOnlyField**
Raised when attempting to update a read-only field.

exception tastytopping.exceptions.**ResourceDeleted**
Raised when attempting to use a deleted resource.

exception tastytopping.exceptions.**ResourceHasNoUri**
Raised when trying to use a not-yet-created Resource's uri().

This can almost always be solved by saving the Resource first.

exception tastytopping.exceptions.**RestMethodNotAllowed**
Raised when the API does not allow a certain REST method (get/post/put/delete).

CHAPTER 2

Requirements

The following needs to be installed locally to run TastyTopping:

- Python 2.7+ or Python 3.3+
- requests >= 1.2.3

Tested with / against:

- django >= 1.5.0
- django-tastypie >= 0.9.14
- requests >= 1.2.3

(see the tox.ini file for more information).

CHAPTER 3

Justification

Why another one? There are some other packages around that do something similar, but none are the complete package:

- ORM. A lot of other packages use a C-style API, which involves passing a dict with your data to their functions. TastyTopping wraps it all up in an ORM-style object, which is more OO, more elegant, and more pythonic.
- Python3 support.
- Support for authentication.
- Support for nested resources.
- QuerySets!
- A thorough set of unit tests.
- Development has stagnated (none of them have released in close to a year, whereas tastypie has been releasing thick and fast).
- Creating this was FUN!

CHAPTER 4

Examples

The examples shown here relate to the following TastyPie Resources:

```
class UserResource(ModelResource):
    class Meta:
        resource_name = 'user'
        queryset = User.objects.all()
        allowed_methods = ['get']
        authorization = Authorization()
        filtering = {
            'username': ALL,
            'id': ALL,
        }

class ExampleResource(models.ModelResource):
    created_by = fields.ForeignKey(UserResource, 'created_by', null=True)
    class Meta:
        resource_name = 'example'
        queryset = Example.objects.all()
        list_allowed_methods   = ['get', 'post']
        detail_allowed_methods = ['get', 'post', 'put', 'delete']
        authentication = ApiKeyAuthentication()
        authorization = Authorization()
        filtering = {
            'title': ALL,
            'rating': ALL,
            'date': ALL,
            'created_by': ALL_WITH_RELATIONS,
        }
        ordering = ['rating', 'date']
```

The following example shows basic usage of the ORM, that will use GET, PUT, POST, and DELETE methods on the API, using the *ResourceFactory*

```
from datetime import datetime
from tastytopping import ResourceFactory, HTTPApiKeyAuth

if __name__ == "__main__":

    factory = ResourceFactory('http://example.api.com:666/test/api/v1/')
    auth = HTTPApiKeyAuth('username', '35632435657adf786c876e097f')
    factory.example.auth = auth

    new_resource = factory.example(title='A Title', rating=50)
```

```
    new_resource.date = datetime.now()
    new_resource.save()

    # Get any user from the list of users and set it to created_by:
    user = factory.user.all().first()
    new_resource.created_by = user
    # Get the new resource by its title:
    another_resource = factory.example.get(title='A Title')
    # Delete the new resource:
    new_resource.delete()
    # This will raise an exception since it's been deleted.
    print another_resource.date
```

CHAPTER 5

Running The Tests

To install tastytopping:

```
git clone https://github.com/cboelsen/tastytopping
cd tastytopping
virtualenv env
. env/bin/activate  # Or, on windows, env/Scripts/activate
pip install -U -r requirements.txt
```

And to run the tests:

```
# Continued from above
pip install tox
tox
```

The tests are run against several environments with different versions of the same packages, and are meant to pass all the tests at all times. If they aren't passing, it's a bug! The tests aren't run against every combination of requests, django, and tastypie supported, though, so there's a small chance a bug might slip in unnoticed.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

t

O

P

Q

R

S

T

U