ElasticUtils Documentation

Release 0.6

Mozilla Foundation

January 17, 2013

CONTENTS

ElasticUtils is a Python library that gives you a Django queryset-like API for elasticsearch as well as some other tools for making it easier to integrate elasticsearch into your application.

Version 0.6

Code https://github.com/mozilla/elasticutils

License BSD; see LICENSE file

Issues https://github.com/mozilla/elasticutils/issues

Documentation http://elasticutils.readthedocs.org/

IRC #elasticutils on irc.mozilla.org

CHAPTER

ONE

USER'S GUIDE

1.1 What's new in ElasticUtils

1.1.1 Version 0.6: Released January 17th, 2013

API-breaking changes:

- **S.values_dict no longer always includes id.**

  `values_dict` no longer always includes an 'id' field in the fields list if you don't specify it.

  Specifying no fields now returns **all** fields:

  ```
  S().values_dict()
  ```

  Specifying fields now returns only those fields:

  ```
  S().values_dict('name', 'number')
  ```

- **S.values_list no longer always includes id.**

  `values_list` no longer always includes an 'id' field in the fields list if you don't specify it.

  Specifying no fields now returns **all** fields:

  ```
  S().values_list()
  ```

  Specifying fields now returns data for those fields in the order the fields are specified:

  ```
  S().values_list('name', 'number')
  ```

- **Types have changed.**

  This is a big change.

  Up through ElasticUtils v0.5, *S* could take a type and that type was a model. This is now completely different.

In ElasticUtils v0.6 and later, *S* takes a *MappingType*. A *MappingType* can be related to a model, but it itself should not be a model. This allows us to return search results as a list of *MappingType* instances which can do things rather than forcing you to do a db hit to get back instances that can do things.

This is similar to how django-haystack works with the SearchIndex class, except ElasticUtils doesn't yet support declarative mapping definition.

See documentation for more details.

- **By default, results are now DefaultMappingType.**

  In ElasticUtils v0.4 and v0.5, if the *S* was untyped and you didn't specify either `values_dict` or `values_list`, then the results would come back as a list of dicts.

  In ElasticUtils v0.5, if the *S* is untyped and you didn't specify either `values_dict` or `values_list`, then the results would come back as a list of *DefaultMappingType*.

  See documentation for more details.

- **elasticutils.contrib.django.models.SearchMixin is no more.**

  The *SearchMixin* class is replaced by *DjangoMappingType* which relates ElasticSearch mapping types to Django ORM models and *Indexable* which is a mixin that adds a bunch of index-related infrastructure.

Changes:

- Added `_source` and `_id` to the metadata decorated on the search results.

  See documentation for more details.

- Fixed `elasticutils.contrib.django.es_required_or_50x`.

  It works better now.

- **prefix filter support.**

  ElasticUtils supports prefix filters. You can do this now:

  ```
  S().filter(name__prefix='odin')
  ```

1.1.2 Version 0.5: Released September 4th, 2012

API-breaking changes:

None.

Changes:

- Added `demote` transform: it adds boosting query support allowing you to do a negative query which reduces scores for documents that match.
- The elasticutils version is now available in `elasticutils.__version__` as well as `elasticutils._version.__version__`.
- Added `__in` support for queries. Doing:

  ```
  S().query(foo__in=['a', 'b', 'c'])
  ```

  does a terms query now.

- Added *MLT* class which does morelikethis.
- Added API documentation for S, an index, `order_by` docs, fixed some icky bugs, and generally improved everything at least a little bit.

1.1.3 Version 0.4: Released July 31st, 2012

API-breaking changes:

- **ElasticUtils no longer requires Django.**

  If you're using Django, you should change your import statements from things like:

  ```
  from elasticutils import get_es, S, F
  ```

  to:

  ```
  from elasticutils.contrib.django import get_es, S, F
  ```

  Further, Django helper modules like `cron`, `tasks`, and `models` were all moved to `elasticutils.contrib.django`.

  We moved *ESTestCase* from `elasticutils.tests` to `elasticutils.contrib.django.estestcase`

  If you don't use Django, ElasticUtils is easier to use!

- **S no longer requires a type.**

  If you're not using Django, *S* no longer requires a type. If you don't specify a type, then ElasticUtils will return results as dicts.

- **Values and values_list changed.**

  `values()` was renamed to `values_list()`.

  `values_list()` (was `values()`) now always returns a list of tuples even if you only requested a single field. Previously, doing something like:

  ```
  searcher = S().values_list('id')
  ```

  would return something like:

  ```
  [1, 2, 3, 4, 5]
  ```

  Now it returns:

  ```
  [(1,), (2,), (3,), (4,), (5,)]
  ```

- **Facet functionality was rewritten.**

  Changed `.facet()` to be arg-driven and allow for *filtered* and *global_* flags.

  Changed `.facets()` to `.facet_counts()` to match Django Haystack.

  Added `.facet_raw()` which allows you to do more complicated facets including scripting. This is similar to the original `.facet()` implementation.

Changes:

- Overhauled and cleaned up ElasticUtils tests. Running tests can be done with:

  ```
  DJANGO_SETTINGS_MODULE=es_settings nosetests
  ```

- Default timeout was changed from 1 second to 5 seconds.
- Added `es` transform: it allows you to specify the settings with which to create an ES when the search is executed.
- Added `es_builder` transform: it allows you to specify a function that builds an ES which will be executed to create an ES when the search is executed.
- Added `indexes` transform: it allows you to specify the indexes to use for the search.

- Added `doctypes` transform: it allows you to specify the doctypes to use for the search.
- Added `explain` transform: it allows you to set the "explain" flag which gives you an explanation of how the score was calculated.

  I also added `elasticutils.utils.format_elasticutils` which formats the resulting explanation text into something slightly more readable. But it's likely this will change in the future.
- Added `boost` transform: it allows you to do query-time field boosting.
- Added support for `prefix`. It's the same as `startswith`, but it uses the same word that ElasticSearch uses. At some point, we'll remove support for `startswith`.
- Added support for `text_phrase` and `query_string` queries.
- Added `highlight` transform: generates highlighted fragments of content that matched the query.
- Removed requirement for nuggets.
- Continued to improve documentation.

1.1.4 Version 0.3: Released June 1st, 2012

Changes:

- Add documentation for debugging, project details and other things.
- Minor project cleanup to make it easier to maintain and use
- Make `get_es()` more useful. It now takes overrides that allow you to configure multiple kinds of ES objects for different purposes.

1.2 Installation

> **Warning:** ElasticUtils doesn't work well with ElasticSearch 0.19.9. If you're using 0.19.9, you should update to at least 0.19.10.
> **https://github.com/elasticsearch/elasticsearch/issues/2205** ElasticSearch bug with `_all`.

There are a few ways to install ElasticUtils.

1.2.1 From PyPI

Do:

```
$ pip install elasticutils
```

1.2.2 From git

Do:

```
$ git clone git://github.com/mozilla/elasticutils.git
```

For other ways to clone, see https://github.com/mozilla/elasticutils.

1.3 Searching with S

1.3.1 Overview

ElasticUtils makes querying and filtering and collecting facets from ElasticSearch simple.

For example:

```
q = (S().filter(product='firefox')
        .filter(version='4.0', platform='all')
        .facet(products={'field': 'product', 'global': True})
        .facet(versions={'field': 'version'})
        .facet(platforms={'field': 'platform'})
        .facet(types={'field': 'type'})
        .doctypes('addon')
        .indexes('addon_index')
        .query(title='Example'))
```

The ElasticSearch REST API curl would look like this:

```
$ curl -XGET 'http://localhost:9200/addon_index/addon/_search' -d '{
'query': {'term': {'title': 'Example'}},
'filter': {'and': [{'term': {'product': 'firefox'}},
                   {'term': {'platform': 'all'}},
```

```
                    {'term': {'version': '4.0'}}]},
'facets': {
   'platforms': {
       'facet_filter': {
           'and': [
               {'term': {'product': 'firefox'}},
               {'term': {'platform': 'all'}},
               {'term': {'version': '4.0'}}]},
       'field': 'platform'},
   'products': {
       'facet_filter': {
           'and': [
               {'term': {'product': 'firefox'}},
               {'term': {'platform': 'all'}},
               {'term': {'version': '4.0'}}]},
       'field': 'product',
       'global': True},
   'types': {
       'facet_filter': {
           'and': [
               {'term': {'product': 'firefox'}},
               {'term': {'platform': 'all'}},
               {'term': {'version': '4.0'}}]},
       'field': 'type'},
   'versions': {
       'facet_filter': {
           'and': [
               {'term': {'product': 'firefox'}},
               {'term': {'platform': 'all'}},
               {'term': {'version': '4.0'}}]},
       'field': 'version'}}
}
'
```

That's it!

For the rest of this chapter, when we translate ElasticUtils queries to their equivalent ElasticSearch REST API, we're going to use a shorthand and only talk about the body of the request which we'll call the *ElasticSearch JSON*.

See Also:

http://www.elasticsearch.org/guide/reference/api/ ElasticSearch docs on api

http://www.elasticsearch.org/guide/reference/api/search/ ElasticSearch docs on search api

http://curl.haxx.se/ Documentation on curl

1.3.2 All about S

Basic untyped S

S is the class that you instantiate to create a search. For example:

```
searcher = S()
```

S has a bunch of methods that all return a new *S* with additional accumulated search criteria.

For example:

```
s1 = S()

s2 = s1.query(content__text='tabs')

s3 = s2.filter(awesome=True)

s4 = s2.filter(awesome=False)
```

s1, *s2*, and *s3* are all different *S* objects. *s1* is a match all.

s2 has a query.

s3 has everything in *s2* plus a `awesome=True` filter.

s4 has everything in *s2* with a `awesome=False` filter.

When you create an *S* with no type, it's called an "untyped S". If you don't specify `.values_dict` or `.values_list`, then your search results are in the form of a sequence of *DefaultMappingType* instances. More about this in *Mapping types*.

Typed S

You can also construct a *typed S* which is an *S* with a *MappingType* subclass. For example:

```
from elasticutils import MappingType, S

class MyMappingType(MappingType):
    @classmethod
    def get_index(cls):
        return 'sumo_index'

    @classmethod
    def get_mapping_type_name(cls):
        return 'mymappingtype'


results = S(MyMappingType).query(title__text='plugins')
```

`results` will be an iterable of *MyMappingType* instances—one for each search result.

1.3.3 Match All

By default `S()` with no filters or queries specified will do a `match_all` query in ElasticSearch.

See Also:

http://www.elasticsearch.org/guide/reference/query-dsl/match-all-query.html ElasticSearch match_all documentation

1.3.4 Queries vs. Filters

A search can contain multiple queries and multiple filters. The two things are very different.

A filter determines whether a document is in the results set or not. If you do a term filter on whether field *foo* has value *bar*, then the result set ONLY has documents where *foo* has value *bar*. Filters are fast and filter results are cached in ElasticSearch when appropriate.

A query affects the score for a document. If you do a term query on whether field *foo* has value *bar*, then the result set will score documents where the query holds true higher than documents where the query does not hold true. Queries are slower than filters and query results are not cached in ElasticSearch.

The other place where this affects things is when you specify facets. See *Facets* for details.

See Also:

http://www.elasticsearch.org/guide/reference/query-dsl/ ElasticSearch Filters and Caching notes

1.3.5 Queries

The query is specified by keyword arguments to the `query()` method. The key of the keyword argument is parsed splitting on ___ (that's two underscores) with the first part as the "field" and the second part as the "field action".

For example:

```
q = S().query(title='taco trucks')
```

will do an elasticsearch term query for "taco trucks" in the title field.

And:

```
q = S().query(title__text='taco trucks')
```

will do a text query instead of a term query.

There are many different field actions to choose from:

field action	elasticsearch query
(no action specified)	term query
term	term query
text	text query
prefix	prefix query [1]
gt, gte, lt, lte	range query
fuzzy	fuzzy query
text_phrase	text_phrase query
query_string	query_string query [2]

See Also:

http://www.elasticsearch.org/guide/reference/query-dsl/ ElasticSearch docs for query dsl

http://www.elasticsearch.org/guide/reference/query-dsl/term-query.html ElasticSearch docs on term queries

http://www.elasticsearch.org/guide/reference/query-dsl/text-query.html ElasticSearch docs on text and text_phrase queries

http://www.elasticsearch.org/guide/reference/query-dsl/prefix-query.html ElasticSearch docs on prefix queries

http://www.elasticsearch.org/guide/reference/query-dsl/range-query.html ElasticSearch docs on range queries

http://www.elasticsearch.org/guide/reference/query-dsl/fuzzy-query.html ElasticSearch docs on fuzzy queries

http://www.elasticsearch.org/guide/reference/query-dsl/query-string-query.html ElasticSearch docs on query_string queries

[1] You can also use `startswith`, but that's deprecated.

[2] When doing `query_string` queries, if the query text is malformed it'll raise a *SearchPhaseExecutionException:* exception.

1.3.6 Filters

```
q = (S().query(title='taco trucks')
        .filter(style='korean'))
```

will do a query for "taco trucks" in the title field and filter on the style field for 'korean'. This is how we find Korean Taco Trucks.

As with `query()`, `filter()` allow for you to specify field actions for the filters:

field action	elasticsearch filter
in	Terms filter
gt, gte, lt, lte	Range filter
prefix, startswith	Prefix filter
(no action)	Term filter

See Also:

http://www.elasticsearch.org/guide/reference/query-dsl/ ElasticSearch docs for query dsl

http://www.elasticsearch.org/guide/reference/query-dsl/terms-filter.html ElasticSearch docs for terms filter

http://www.elasticsearch.org/guide/reference/query-dsl/range-filter.html ElasticSearch docs for range filter

http://www.elasticsearch.org/guide/reference/query-dsl/prefix-filter.html ElasticSearch docs for prefix filter

http://www.elasticsearch.org/guide/reference/query-dsl/term-filter.html ElasticSearch docs for term filter

1.3.7 Advanced filters and F

Calling filter multiple times is equivalent to an "and"ing of the filters.

For example:

```
q = (S().filter(style='korean')
        .filter(price='FREE'))
```

will do a query for style 'korean' AND price 'FREE'. Anything that has a style other than 'korean' or a price other than 'FREE' is removed from the result set.

This translates to:

```
{'filter': {
    'and': [
        {'term': {'style': 'korean'}},
        {'term': {'price': 'FREE'}}
    ]}
}
```

in elasticutils JSON.

You can do the same thing by putting both filters in the same `.filter()` call.

For example:

```
q = S().filter(style='korean', price='FREE')
```

that also translates to:

```
{'filter': {
    'and': [
        {'term': {'style': 'korean'}},
        {'term': {'price': 'FREE'}}
    ]}
}
```

in elasticutils JSON.

Suppose you want either Korean or Mexican food. For that, you need an "or".

You can do something like this:

```
q = S().filter(or_={'style': 'korean', 'style'='mexican'})
```

That translates to:

```
{'filter': {
    'or': [
        {'term': {'style': 'korean'}},
        {'term': {'style': 'mexican'}}
    ]}
}
```

But, that's kind of icky looking.

So, we've also got an `F` class that makes this sort of thing easier.

You can do the previous example with `F` like this:

```
q = S().filter(F(style='korean') | F(style='mexican'))
```

will get you all the search results that are either "korean" or "mexican" style.

That translates to:

```
{'filter': {
    'or': [
        {'term': {'style': 'korean'}},
        {'term': {'style': 'mexican'}}
    ]}
}
```

What if you want Mexican food, but only if it's FREE, otherwise you want Korean?:

```
q = S().filter(F(style='mexican', price='FREE') | F(style='korean'))
```

That translates to:

```
{'filter': {
    'or': [
        {'and': [
            {'term': {'price': 'FREE'}},
            {'term': {'style': 'mexican'}}
        ]},
        {'term': {'style': 'korean'}}
    ]}
}
```

`F` supports AND, OR, and NOT operators which are `&`, `|` and `~` respectively.

Additionally, you can create an empty `F` and build it incrementally:

```
qs = S()
f = F()
if some_crazy_thing:
    f &= F(price='FREE')
if some_other_crazy_thing:
    f |= F(style='mexican')

qs = qs.filter(f)
```

If neither *some_crazy_thing* or *some_other_crazy_thing* are `True`, then `F` will be empty. That's ok because empty filters are ignored.

1.3.8 Query-time field boosting

ElasticSearch allows you to boost scores for fields specified in the search query at query-time.

ElasticUtils allows you to specify query-time field boosts with `.boost()`. It takes a set of arguments where the keys are either field names or field name + '__' + field action.

Here's an example:

```
q = (S().query(title='taco trucks',
               description__text='awesome')
        .boost(title=4.0, description__text=2.0))
```

If the key is a field name, then the boost will apply to all query bits that have that field name. For example:

```
q = (S().query(title='trucks',
               title__prefix='trucks',
               title__fuzzy='trucks')
        .boost(title=4.0))
```

applies a 4.0 boost to all three query bits because all three query bits are for the `title` field name.

If the key is a field name and field action, then the boost will apply only to that field name and field action. For example:

```
q = (S().query(title='trucks',
               title__prefix='trucks',
               title__fuzzy='trucks')
        .boost(title__prefix=4.0))
```

will only apply the 4.0 boost to `title__prefix`.

1.3.9 Ordering

You can order search results by specified fields:

```
q = (S().query(title='trucks')
        .order_by('title')
```

This orders search results by the *title* field in ascending order.

If you want to sort by descending order, prepend a `-`:

```
q = (S().query(title='trucks')
        .order_by('-title')
```

You can also sort by the computed field `_score`.

See Also:

http://www.elasticsearch.org/guide/reference/api/search/sort.html ElasticSearch docs on sort parameter in the Search API

1.3.10 Demoting

You can demote documents that match query criteria:

```
q = (S().query(title='trucks')
        .demote(0.5, description__text='gross'))
```

This does a query for trucks, but demotes any that have "gross" in the description with a fraction boost of 0.5.

Note: You can only call `.demote()` once. Calling it again overwrites previous calls.

This is implemented using the *boosting query* in ElasticSearch. Anything you specify with `.query()` goes into the *positive* section. The *negative query* and *negative boost* portions are specified as the first and second arguments to `.demote()`.

Note: Order doesn't matter. So:

```
q = (S().query(title='trucks')
        .demote(0.5, description__text='gross'))
```

does the same thing as:

```
q = (S().demote(0.5, description__text='gross')
        .query(title='trucks'))
```

See Also:

http://www.elasticsearch.org/guide/reference/query-dsl/boosting-query.html ElasticSearch docs on boosting query (which are as clear as mud)

1.3.11 Highlighting

ElasticUtils allows you to highlight excerpts that match the query using the `.highlight()` transform. This returns data that will be in every item in the search results list as `_highlight`.

For example, let's do a query on a search corpus of knowledge base articles for articles with the word "crash" in them:

```
q = (S().query(title__text='crash', content__text='crash')
        .highlight('title', 'content'))

for result in q:
    print result._highlight['title']
    print result._highlight['content']
```

This will print two lists. The first is highlighted fragments from the title field. The second is highlighted fragments from the content field.

Highlighting is done in ElasticSearch and covers all the query bits. So if you had a document like this:

```
{
    "title": "How not to be seen",
    "content": "The first rule of how not to be seen: don't stand up."
}
```

And did this query:

```
q = (S().query(title__text="rule seen", content__text="rule seen")
        .highlight('title', 'content'))
```

Then the highlights you'd get back would be:

- title: `to be <em>seen</em>`
- content: `first <em>rule</em> of how not to be <em>seen</em>:  don't stand up.`

The "highlight" default is to wrap the matched text with `<em>` and `</em>`. You can change this by passing in `pre_tags` and `post_tags` options:

```
q = (S().query(title__text='crash', content__text='crash')
        .highlight('title', 'content',
                   pre_tags=['<b>'],
                   post_tags=['</b>']))
```

If you need to clear the highlight, call `.highlight()` with `None`. For example, this search won't highlight anything:

```
q = (S().query(title__text='crash')
        .highlight('title')          # highlights 'title' field
        .highlight(None))            # clears highlight
```

Note: Make sure the fields you're highlighting are indexed correctly. Check the ElasticSearch documentation for details.

See Also:

http://www.elasticsearch.org/guide/reference/api/search/highlighting.html ElasticSearch docs for highlight

1.3.12 Facets

Basic facets

```
q = (S().query(title='taco trucks')
        .facet('style', 'location'))
```

will do a query for "taco trucks" and return terms facets for the `style` and `location` fields.

That translates to:

```
{'query': {'term': {'title': 'taco trucks'}},
 'facets': {
     'style': {'terms': {'field': 'style'}},
     'location': {'terms': {'field': 'location'}}
     }
}
```

Note that the fieldname you provide in the `.facet()` call becomes the facet name as well.

The facet counts are available through `.facet_counts()` on the *S* instance. For example:

```
q = (S().query(title='taco trucks')
        .facet('style', 'location'))
counts = q.facet_counts()
```

See Also:

http://www.elasticsearch.org/guide/reference/api/search/facets/ ElasticSearch docs on facets

http://www.elasticsearch.org/guide/reference/api/search/facets/terms-facet.html ElasticSearch docs on terms facet

Facets and scope (filters and global)

What happens if your search includes filters?

Here's an example:

```
q = (S().query(title='taco trucks')
        .filter(style='korean')
        .facet('style', 'location'))
```

That translates to this:

```
{'query': {'term': {'title': 'taco trucks'}},
 'filter': {'term': {'style': 'korean'}},
 'facets': {
     'style': {
         'terms': {'field': 'style'}
         },
     'location': {
         'terms': {'field': 'location'}
         }
     }
 }
```

The "style" and "location" facets here ONLY apply to the results of the query and are not affected at all by the filters.

If you want your filters to apply to your facets as well, pass in the filtered flag:

```
q = (S().query(title='taco trucks')
        .filter(style='korean')
        .facet('style', 'location', filtered=True))
```

That translates to this:

```
{'query': {'term': {'title': 'taco trucks'}},
 'filter': {'term': {'style': 'korean'}},
 'facets': {
     'styles': {
         'facet_filter': {'term': {'style': 'korean'}},
         'terms': {'field': 'style'}
         },
     'locations': {
         'facet_filter': {'term': {'style': 'korean'}},
         'terms': {'field': 'location'}
         }
     }
}
```

Notice how there's an additional *facet_filter* component to the facets and it contains the contents of the original *filter* component.

What if you want the filters to apply just to one of the facets and not the other? You need to add them incrementally:

```
q = (S().query(title='taco trucks')
        .filter(style='korean')
        .facet('style', filtered=True)
        .facet('location'))
```

That translates to this:

```
{'query': {'term': {'title': 'taco trucks'}},
 'filter': {'term': {'style': 'korean'}},
 'facets': {
     'style': {
         'facet_filter': {'term': {'style': 'korean'}},
         'terms': {'field': 'style'}
         },
     'location': {
         'terms': {'field': 'location'}
         }
     }
 }
```

What if you want the facets to apply to the entire corpus and not just the results from the query? Use the *global_* flag:

```
q = (S().query(title='taco trucks')
        .filter(style='korean')
        .facet('style', 'location', global_=True))
```

That translates to this:

```
{'query': {'term': {'title': 'taco trucks'}},
 'filter': {'term': {'style': 'korean'}},
 'facets': {
    'style': {
         'global': True,
         'terms': {'field': 'style'}
         },
    'location': {
         'global': True,
         'terms': {'field': 'location'}
         }
    }
}
```

Note: The flag name is *global_* with an underscore at the end. Why? Because *global* with no underscore is a Python keyword.

See Also:

http://www.elasticsearch.org/guide/reference/api/search/facets/ ElasticSearch docs on facets, facet_filter, and global

http://www.elasticsearch.org/guide/reference/api/search/facets/terms-facet.html ElasticSearch docs on terms facet

Facets... RAW!

ElasticSearch facets can do a lot of other things. Because of this, there exists `.facet_raw()` which will do whatever you need it to. Specify key/value args by facet name.

For example, you can do the first facet example by:

```
q = (S().query(title='taco trucks')
        .facet_raw(style={'terms': {'field': 'style'}}))
```

One of the things this lets you do is scripted facets. For example:

```
q = (S().query(title='taco trucks')
        .facet_raw(styles={
            'field': 'style',
            'script': 'term == korean ? true : false'
        }))
```

That translates to:

```
{'query': {'term': {'title': 'taco trucks'}},
 'facets': {
     'styles': {
         'field': 'style',
         'script': 'term == korean ? true : false'
         }
     }
}
```

Warning: If for some reason you have specified a facet with the same name using both `.facet()` and `.facet_raw()`, the `.facet_raw()` one will override the `.facet()` one.

See Also:

http://www.elasticsearch.org/guide/reference/modules/scripting.html ElasticSearch docs on scripting

1.3.13 Counts

Total hits can be found by using `.count()`. For example:

```
q = S().query(title='taco trucks')
count = q.count()
```

Note: Don't use Python's `len` built-in on the *S* instance if you want the number of documents in your index that matches your search.

This:

```
q = S()
...
q.count()
```

asks ElasticSearch how many documents in the index match your search.

This:

```
q = S()
...
len(q)
```

performs the search, gets back as many documents as specified by the limits of your search, and returns the length of that list of documents.

1.3.14 Mapping types

MappingType lets you specify the instance type for search results you get back from ElasticSearch searches. You can additionally relate a *MappingType* to a database model allowing you to link documents in the ElasticSearch index back to database objects in a lazy-loading way.

Creating a *MappingType* lets you specify the index and doctype easily. It also lets you tie business logic to your search results.

For example, say you had a description field and wanted to have a truncated version of it:

```
class MyMappingType(MappingType):
    def description_truncated(self):
        return self.description[:100]

res = list(S(MyMappingType).query(description__text='stormy night'))[0]

print res.description_truncated()
```

The most basic *MappingType* is the *DefaultMappingType* which is returned if you don't specify a *MappingType* and also don't specify `values_dict` or `values_list`. The *DefaultMappingType* lets you access search result fields as instance attributes or as keys:

```
res.description
res['description']
```

The latter syntax is helpful when there are attributes defined on the class that have the same name as the document field.

To create a *MappingType* you should probably override at least *get_index* and *get_mapping_type_name*. If you want to tie the *MappingType* to a database model, then you should define *get_model* which relates the *MappingType* to a database model class and *get_object* which returns the database object related to that search result. For example:

```
class ContactType(MappingType):
    @classmethod
    def get_index(cls):
        return 'contacts_index'

    @classmethod
    def get_mapping_type_name(cls):
        return 'contact_type'

    @classmethod
    def get_model(cls):
        return ContactModel

    def get_object(self):
        return self.get_model().get(id=self._id)
```

1.3.15 Results

By default

Results are lazy-loaded, so the query will not be made until you try to access an item or some other attribute requiring the data.

If you have a typed *S* (e.g. `S(MappingType)`), then by default, results will be instances of that type.

If you have an untyped *S* (e.g. `S()`), then by default, results will be *DefaultMappingType*.

Results as a list of tuples

values_list with no arguments returns a list of tuples of all the data for that document. With arguments, it'll return a list of tuples of values of the fields specified in the order the fields were specified.

For example:

```
>>> list(S().values_list())
[(1, 'fred', 40), (2, 'brian', 30), (3, 'james', 45)]
>>> list(S().values_list('id', 'name'))
[(1, 'fred'), (2, 'brian'), (3, 'james')]
>>> list(S().values_list('name', 'id')
[('fred', 1), ('brian', 2), ('james', 3)]
```

Note: If you don't specify fields, the data comes back in an arbitrary order. It's probably best to specify fields or use `values_dict`.

Results as a list of dicts

values_dict returns a list of dicts. With no arguments, it returns a list of dicts with all the fields. With arguments, it returns a list of dicts with specified fields.

For example:

```
>>> list(S().values_dict())
[{'id': 1, 'name': 'fred', 'age': 40}, {'id': 2, 'name': 'dennis', 'age': 37}]
>>> list(S().values_dict('id', 'name')
[{'id': 1, 'name': 'fred'}, {'id': 2, 'name': 'brian'}]
```

1.3.16 Scores and explanations

Seeing the score

Wondering what the score for a document was? ElasticUtils puts that in the `_score` on the search result. For example, let's search an index that holds knowledge base articles for ones with the word "crash" in them and print out the scores:

```
q = S().query(title__text='crash', content__text='crash')

for result in q:
    print result._score
```

This works regardless of what form the search results are in.

Getting an explanation

Wondering why one document shows up higher in the results than another that should have shown up higher? Wonder how that score was computed? You can set the search to pass the `explain` flag to ElasticSearch with the `.explain()` transform.

This returns data that will be in every item in the search results list as `_explanation`.

For example, let's do a query on a search corpus of knowledge base articles for articles with the word "crash" in them:

```
q = (S().query(title__text='crash', content__text='crash')
        .explain())

for result in q:
    print result._explanation
```

This works regardless of what form the search results are in.

See Also:

http://www.elasticsearch.org/guide/reference/api/search/explain.html ElasticSearch docs on explain (which are pretty bereft of details).

1.3.17 API

The S class

class `elasticutils.`**`S`**(*type_=None*)

Represents a lazy ElasticSearch Search API request.

The API for *S* takes inspiration from Django's QuerySet.

S can be either typed or untyped. An untyped *S* returns dict results by default.

An *S* is lazy in the sense that it doesn't do an ElasticSearch search request until it's forced to evaluate by either iterating over it, calling `.count`, doing `len(s)`, or calling `.facet_count`.

`__init__`(*type_=None*)

Create and return an S.

Parameters **type** – class; the model that this S is based on

Chaining transforms

`query`(***kw*)

Return a new S instance with query args combined with existing set.

`filter`(**filters*, ***kw*)

Return a new S instance with filter args combined with existing set.

`order_by`(**fields*)

Return a new S instance with the ordering changed.

`boost`(***kw*)

Return a new S instance with field boosts.

`demote`(*amount_*, ***kw*)

Returns a new S instance with boosting query and demotion.

`facet`(**args*, ***kw*)

Return a new S instance with facet args combined with existing set.

facet_raw(***kw*)
Return a new S instance with raw facet args combined with existing set.

highlight(**fields*, ***kwargs*)
Set highlight/excerpting with specified options.

This highlight will override previous highlights.

This won't let you clear it–we'd need to write a `clear_highlight()`.
Parameters fields – The list of fields to highlight. If the field is None, then the highlight is cleared.
Additional keyword options:
- `pre_tags` – List of tags before highlighted portion
- `post_tags` – List of tags after highlighted portion

See ElasticSearch highlight:

http://www.elasticsearch.org/guide/reference/api/search/highlighting.html

values_list(**fields*)
Return a new S instance that returns ListSearchResults.
Parameters fields – the list of fields to have in the results. By default this is at least `['id']`.

values_dict(**fields*)
Return a new S instance that returns DictSearchResults.
Parameters fields – the list of fields to have in the results. By default, this won't specify fields and thus ES will return everything.

es(***settings*)
Return a new S with specified ES settings.

This allows you to configure the ES that gets used to execute the search.
Parameters settings – the settings you'd use to build the ES—same as what you'd pass to `get_es()`.

es_builder(*builder_function*)
Return a new S with specified ES builder.

When you do something with an S that causes it to execute a search, then it will call the specified builder function with the S instance. The builder function will return an ES object that the S will use to execute the search with.
Parameters builder_function – function; takes an S instance and returns an ES
This is handy for caching ES instances. For example, you could create a builder that caches ES instances thread-local:

```
from threading import local
_local = local()

def thread_local_builder(searcher):
    if not hasattr(_local, 'es'):
        _local.es = get_es()
    return _local.es

searcher = S.es_builder(thread_local_builder)
```

This is also handy for building ES instances with configuration defined in a config file.

indexes(**indexes*)
Return a new S instance that will search specified indexes.

doctypes(**doctypes*)
Return a new S instance that will search specified doctypes.

Note: ElasticSearch calls these "mapping types". It's the name associated with a mapping.

---

explain(*value=True*)
Return a new S instance with explain set.

Methods to override if you need different behavior

get_es(*default_builder=<function get_es at 0x2986758>*)
Returns the ES object to use.

get_indexes(*default_indexes=['default']*)
Returns the list of indexes to act on.

get_doctypes(*default_doctypes=['document']*)
Returns the list of doctypes to use.

Methods that force evaluation

count()
Return the number of hits for the search as an integer.

facet_counts()

1.4 More like this with MLT

1.4.1 Overview

ElasticUtils exposes ElasticSearch More Like This API with the *MLT* class.

For example:

```
mlt = MLT(2034, index='addon_index', doctype='addon')
```

This creates an *MLT* that will return documents that are like document 2034 of type *addon* in the *addon_index*.

You can specify an *S* and the *MLT* will derive the index, doctype, ES object, and also use the search specified by the S in the body of the More Like This request. This allows you to get documents like the one specified that also meet query and filter criteria. For example:

```
s = S().filter(product='firefox')
mlt = MLT(2034, s=s)
```

You can specify which fields to use with the *fields* argument. If you don't, then ElasticSearch will use all the fields.

You can specify additional parameters. See the documentation on the moreLikeThis query.

See Also:

http://www.elasticsearch.org/guide/reference/api/more-like-this.html ElasticSearch guide on More Like This API

http://www.elasticsearch.org/guide/reference/query-dsl/mlt-query.html ElasticSearch guide on the moreLikeThis query which specifies the additional parameters you can use.

1.4.2 API

class elasticutils.**MLT**(*id_*, *s=None*, *fields=None*, *index=None*, *doctype=None*, *es=None*, ***query_params*)

Represents a lazy ElasticSearch More Like This API request.

This is lazy in the sense that it doesn't evaluate and execute the ElasticSearch request unless you force it to by iterating over it or getting the length of the search results.

For example:

```
>>> mlt = MLT(2034, index='addons_index', doctype='addon')
>>> num_related_documents = len(mlt)
>>> num_related_documents = list(mlt)
```

__init__(*id_*, *s=None*, *fields=None*, *index=None*, *doctype=None*, *es=None*, ***query_params*)

When the MLT is evaluated, it generates a list of dict results.

Parameters

- **id** – The id of the document we want to find more like.
- **s** – An instance of an S. The query is passed in the body of the more like this request.
- **fields** – A list of fields to use for more like this.
- **index** – The index to use. Falls back to the first index listed in s.
- **doctype** – The doctype to use. Falls back to the first doctype listed in s.
- **es** – The ES object to use. If you don't provide one, then it will create one for you.
- **query_params** – Any additional query parameters for the more like this call.

Note: You must specify either an *s* or the *index* and *doctype* arguments. Omitting them will result in a *ValueError*.

---

get_es()

Returns an ES

•If there's an s, then it returns that ES.

•If the es was provided in the constructor, then it returns that ES.

•Otherwise, it creates a new ES and returns that.

Override this if that behavior isn't correct for you.

raw()

Build query and passes to ElasticSearch, then returns the raw format returned.

1.5 Getting an ES

ElasticUtils uses *pyes* which comes with a handy *ES* object. This lets you work with ElasticSearch outside of what ElasticUtils can do.

To access this, you use *get_es()* which builds an *ES*.

elasticutils.**get_es**(*hosts=None*, *default_indexes=None*, *timeout=None*, *dump_curl=None*, ***settings*)

Create an ES object and return it.

Parameters

- **hosts** – list of uris; ES hosts to connect to, defaults to `['localhost:9200']`
- **default_indexes** – list of strings; the default indexes to use, defaults to 'default'
- **timeout** – int; the timeout in seconds, defaults to 5
- **dump_curl** – function or None; function that dumps curl output, see docs, defaults to None
- **settings** – other settings to pass into *pyes.es.ES*

Examples:

```
>>> es = get_es()

>>> es = get_es(hosts=['localhost:9200'])

>>> es = get_es(timeout=30)  # good for indexing

>>> es = get_es(default_indexes=['sumo_prod_20120627']

>>> class CurlDumper(object):
...     def write(self, text):
...         print text
...
>>> es = get_es(dump_curl=CurlDumper())
```

Warning: ElasticUtils works with `pyes` 0.15 and 0.16. The API for later versions of pyes has changed too much and won't work with ElasticUtils. We're planning to switch to something different in the future.

1.6 Debugging

Here are a few helpful utilities for debugging your ElasticUtils work.

1.6.1 Score explanations

Want to see how a score for a search result was calculated? See *Scores and explanations*.

1.6.2 get_es dump_curl

You can pass a function into *get_es()* which will let you dump the curl equivalents.

For example:

```
from elasticutils import get_es

class CurlDumper(object):
    def write(self, s):
        print s

es = get_es(dump_curl=CurlDumper())
```

1.6.3 elasticsearch-head

https://github.com/mobz/elasticsearch-head

elasticsearch-head is the phpmyadmin for elasticsearch. It makes it much easier to see what's going on.

1.6.4 elasticsearch-paramedic

https://github.com/karmi/elasticsearch-paramedic

elasticsearch-paramedic allows you to see the state and real-time statistics of your ES cluster.

1.6.5 ngrep

http://ngrep.sourceforge.net/

Sometimes, it helps to see exactly what's going over the wire. ngrep has a horrible web-site, but it's a super handy tool for seeing the complete conversation. You can use it like this:

```
$ ngrep -d any -p 9200
```

And then run your program and watch the output.

I often use this when testing sample ElasticUtils programs to see how mappings, document values, facets, filters, queries and all that work.

1.7 elasticutils.contrib.django: Using with Django

1.7.1 Summary

Django helpers are all located in *elasticutils.contrib.django*.

This chapter covers using ElasticUtils Django bits.

1.7.2 Configuration

ElasticUtils depends on the following settings in your Django settings file:

`django.conf.settings.`**`ES_DISABLED`**
If *ES_DISABLED = True*, then Any method wrapped with *es_required* will return and log a warning. This is useful while developing, so you don't have to have ElasticSearch running.

`django.conf.settings.`**`ES_DUMP_CURL`**
If set to a file path all the requests that *ElasticUtils* makes will be dumped into the designated file.

If set to a class instance, calls the `.write()` method with the curl equivalents.

See *Debugging* for more details.

`django.conf.settings.`**`ES_HOSTS`**
This is a list of ES hosts. In development this will look like:

```
ES_HOSTS = ['127.0.0.1:9200']
```

`django.conf.settings.`**`ES_INDEXES`**
This is a mapping of doctypes to indexes. A *default* mapping is required for types that don't have a specific index.

When ElasticUtils queries the index for a model, by default it derives the doctype from *Model._meta.db_table*. When you build your indexes and mapping types, make sure to match the indexes and mapping types you're using.

Example 1:

```
ES_INDEXES = {'default': 'main_index'}
```

This only has a default, so all ElasticUtils queries will look in *main_index* for all mapping types.

Example 2:

```
ES_INDEXES = {'default': 'main_index',
              'splugs': 'splugs_index'}
```

Assuming you have a *Splug* model which has a *Splug._meta.db_table* value of *splugs*, then ElasticUtils will run queries for *Splug* in the *splugs_index*. ElasticUtils will run queries for other models in *main_index* because that's the default.

Example 3:

```
ES_INDEXES = {'default': ['main_index'],
              'splugs': ['splugs_index']}
```

FIXME: The API allows for this. Pretty sure it should query multiple indexes, but we have no tests for that and I haven't tested it, either.

`django.conf.settings.`**`ES_TIMEOUT`**
Defines the timeout for the *ES* connection. This defaults to 5 seconds.

1.7.3 ES

The *get_es()* in the Django contrib will helpfully cache your ES objects thread-local.

It is built with the settings from your *django.conf.settings*.

Note: *get_es()* only caches the *ES* if you don't pass in any override arguments. If you pass in override arguments, it doesn't cache it and instead creates a new one.

1.7.4 Using with Django ORM models

Requirements Django

The *elasticutils.contrib.django.S* class takes a *MappingType* in the constructor. That allows you to tie Django ORM models to ElasticSearch index search results.

In `elasticutils.contrib.django.models` is *DjangoMappingType* which has some additional Django ORM-specific code in it to make it easier.

Define a *DjangoMappingType* subclass for your model. The minimal you need to define is *get_model*.

Further, you can use the *Indexable* mixin to get a bunch of helpful indexing-related code.

For example, here's a minimal *DjangoMappingType* subclass:

```
from django.models import Model
from elasticutils.contrib.django.models import DjangoMappingType


class MyModel(Model):
    ...


class MyMappingType(DjangoMappingType):
    @classmethod
    def get_model(cls):
        return MyModel

searcher = MyMappingType.search()
```

Here's one that uses *Indexable* and handles indexing:

```
from django.models import Model
from elasticutils.contrib.django.models import DjangoMappingType


class MyModel(Model):
    ...


class MyMappingType(DjangoMappingType, Indexable):
    @classmethod
    def get_model(cls):
        return MyModel

    @classmethod
    def extract_document(cls, obj_id, obj=None):
        if obj is None:
            obj = cls.get_model().get(pk=obj_id)

        return {
            'id': obj.id,
            'name': obj.name,
            'bio': obj.bio,
```

```
            'age': obj.age
            }


searcher = MyMappingType.search()
```

This example doesn't specify a mapping. That's ok because ElasticSearch will infer from the shape of the data how it should analyze and store the data.

If you want to specify this explicitly (and I suggest you do for anything that involves strings), then you want to additionally override *.get_mapping()*. Let's refine the above example by explicitly specifying *.get_mapping()*.

```
from django.models import Model
from elasticutils.contrib.django.models import DjangoMappingType


class MyModel(Model):
    ...


class MyMappingType(DjangoMappingType, Indexable):
    @classmethod
    def get_model(cls):
        return MyModel

    @classmethod
    def get_mapping(cls):
        """Returns an ElasticSearch mapping."""
        return {
            # The id is an integer, so store it as such. ES would have
            # inferred this just fine.
            'id': {'type': 'integer'},

            # The name is a name---so we shouldn't analyze it
            # (de-stem, tokenize, parse, etc).
            'name': {'type': 'string', 'index': 'not_analyzed'},

            # The bio has free-form text in it, so analyze it with
            # snowball.
            'bio': {'type': 'string', 'analyzer': 'snowball'},

            # Age is an integer
            'age': {'type': 'integer'}
            }

    @classmethod
    def extract_document(cls, obj_id, obj=None):
        if obj is None:
            obj = cls.get_model().get(pk=obj_id)

        return {
            'id': obj.id,
            'name': obj.name,
            'bio': obj.bio,
            'age': obj.age
            }


searcher = MyMappingType.search()
```

DjangoMappingType

class elasticutils.contrib.django.models.**DjangoMappingType**
This has most of the pieces you need to tie back to a Django ORM model.

Subclass this and override at least *get_model*.

classmethod get_index()
Gets the index for this model.

The index for this model is specified in *settings.ES_INDEXES* which is a dict of mapping type -> index name.

By default, this uses *.get_mapping_type()* to determine the mapping and returns the value in *settings.ES_INDEXES* for that or `settings.ES_INDEXES['default']`.

Override this to compute it differently.

Returns index name to use

classmethod get_mapping_type_name()
Returns the name of the mapping.

By default, this is `cls.get_model()._meta.db_table`.

Override this if you want to compute the mapping type name differently.

Returns mapping type string

classmethod get_model()
Return the model related to this DjangoMappingType.

This can be any class that has an instance related to this DjangoMappingtype by id.

Override this to return a model class.

Returns model class

classmethod search()
Returns a typed S for this class.

Returns an *S*

Indexable

class elasticutils.contrib.django.models.**Indexable**
Mixin for mapping types with all the indexing hoo-hah.

Add this mixin to your DjangoMappingType subclass and it gives you super indexing power.

classmethod extract_document(*obj_id*, *obj=None*)
Extracts the ES index document for this instance

This must be implemented.

Note: The resulting dict must be JSON serializable.

Parameters

- **obj_id** – the object id for the instance to extract from
- **obj** – if this is not None, use this as the object to extract from; this allows you to fetch a bunch of items at once and extract them one at a time

Returns dict of key/value pairs representing the document

classmethod get_indexable()
Returns the queryset of ids of all things to be indexed.

Defaults to:

```
cls.get_model().objects.order_by('id').values_list('id', flat=True)
```

Returns iterable of ids of objects to be indexed

classmethod get_mapping()
Returns the mapping for this mapping type.

See the docs for details on how to specify a mapping.

Override this to return a mapping for this doctype.

Returns dict representing the ES mapping or None if you want ES to infer it. defaults to None.

classmethod index(*document*, *id_=None*, *bulk=False*, *force_insert=False*, *es=None*)
Adds or updates a document to the index

Parameters

- **document** – Python dict of key/value pairs representing the document

  ---

  Note: This must be serializable into JSON.

- **id** – the Django ORM model instance id—this is used to convert an ES search result back to the Django ORM model instance from the db. It should be an integer.

  Note: If you don't provide an `id_`, then ElasticSearch will make up an id for your document and it'll look like a character name from a Lovecraft novel.

- **bulk** – Whether or not this is part of a bulk indexing. If this is, you must provide an ES with the *es* argument, too.
- **force_insert** – TODO
- **es** – The ES to use. If you don't specify an ES, it'll use *elasticutils.contrib.django.get_es()*.

Raises ValueError if *bulk* is True, but *es* is None.

Note: After you add things to the index, make sure to refresh the index by calling `refresh_index()` — it doesn't happen automatically.

TODO: add example.

classmethod refresh_index(*timesleep=0*, *es=None*)
Refreshes the index.

TODO: document this better.

classmethod unindex(*id_*, *es=None*)
Removes a particular item from the search index.

TODO: document this better.

See Also:

http://www.elasticsearch.org/guide/reference/mapping/ The ElasticSearch guide on mapping types.

http://www.elasticsearch.org/guide/reference/mapping/core-types.html The ElasticSearch guide on mapping type field types.

1.7.5 Other helpers

Requirements Django, Celery

You can then utilize things such as `elasticutils.contrib.django.tasks.index_objects()` to automatically index all new items.

View decorators

`elasticutils.contrib.django.`**`es_required`**(*fun*)

Wrap a callable and return None if ES_DISABLED is False.

This also adds an additional *es* argument to the callable giving you an ES to use.

`elasticutils.contrib.django.`**`es_required_or_50x`**(*disabled_template='elasticutils/501.html'*, *error_template='elasticutils/503.html'*)

Wrap a Django view and handle ElasticSearch errors.

This wraps a Django view and returns 501 or 503 status codes and pages if things go awry.

HTTP 501 Returned when `ES_DISABLED` is True.

HTTP 503 Returned when any of the following exceptions are thrown:

- pyes.urllib3.MaxRetryError: Connection problems with ES.
- pyes.exceptions.IndexMissingException: When the index is missing.
- pyes.exceptions.ElasticSearchException: Various other ElasticSearch related errors.

Template variables:

- error: A string version of the exception thrown.

Parameters

- **disabled_template** – The template to use when ES_DISABLED is True.

  Defaults to `elasticutils/501.html`.

- **error_template** – The template to use when ElasticSearch isn't working properly, is missing an index, or something along those lines.

  Defaults to `elasticutils/503.html`.

Examples:

```
# This creates a home_view and decorates it to use the
# default templates.

@es_required_or_50x()
def home_view(request):
    ...


# This creates a search_view and overrides the templates
```

```
@es_required_or_50x(disabled_template='search/es_disabled.html',
                    error_template('search/es_down.html')
def search_view(request):
    ...
```

Tasks

elasticutils.contrib.django.tasks.**index_objects**(*model, ids=[...]*)
Models can asynchronously update their ES index.

If a model extends SearchMixin, it can add a post_save hook like so:

```
@receiver(dbsignals.post_save, sender=MyModel)
def update_search_index(sender, instance, **kw):
    from elasticutils import tasks
    tasks.index_objects.delay(sender, [instance.id])
```

Cron

elasticutils.contrib.django.cron.**reindex_objects**(*model, chunk_size[=150]*)
Creates methods that reindex all the objects in a model.

For example in your `myapp.cron.py` you can do:

```
index_all_mymodels = cronjobs.register(reindex_objects(mymodel))
```

and it will create a commandline callable task for you, e.g.:

```
./manage.py cron index_all_mymodels
```

1.7.6 Writing tests

Requirements Django, test_utils, nose

In *elasticutils.contrib.django.estestcase*, is *ESTestCase* which can be subclassed in your app's test cases.

It does the following:

- If *ES_HOSTS* is empty it raises a *SkipTest*.
- *self.es* is available from the *ESTestCase* class and any subclasses.
- At the end of the test case the index is wiped.

Example:

```
from elasticutils.djangolib import ESTestCase


class TestQueries(ESTestCase):
    def test_query(self):
        ...

    def test_locked_filters(self):
        ...
```

1.7.7 Debugging

You can set the `settings.ES_DUMP_CURL` to a few different things all of which can be helpful in debugging ElasticUtils.

1. a file path

   This will cause PyES to write the curl equivalents of the commands it's sending to ElasticSearch to a file.

   Example setting:

   ```
   ES_DUMP_CURL = '/var/log/es_curl.log'
   ```

   Note: The file is not closed until the process ends. Because of that, you don't see much in the file until it's done.

   ---

2. a class instance that has a `.write()` method

   PyES will call the `.write()` method with the curl equivalent and then you can do whatever you want with it.

   For example, this writes curl equivalent output to stdout:

   ```
   class CurlDumper(object):
       def write(self, s):
           print s
   ES_DUMP_CURL = CurlDumper()
   ```

CHAPTER

TWO

CONTRIBUTOR'S GUIDE

2.1 Join this project!

Interested in working on a Python library for using elasticsearch? Interested in using it? Then you should be interested in this project!

2.1.1 Want to help?

Here are things we need help with:

- **fixing bugs listed in the issue tracker**
- **writing tests**
- **writing documentation**: We could use help writing better documentation for ElasticUtils.
- **spreading the word**: Do you know other people who would like this software? If so, tell them about ElasticUtils!
- **project infrastructure**: Is there infrastructure that's missing in this project that would make it easier for you to collaborate? If so, what?

Are you thinking, "That list is makes me want to go shopping for bumper stickers!" That's ok! Hop on IRC, say hi and we can go from there!

For project details, see *ElasticUtils*.

2.2 Hacking HOWTO

This covers setting up a development environment for developing on ElasticUtils. If you're interested in using ElasticUtils, then you should check out *User's Guide*.

2.2.1 External requirements

You should have elasticsearch installed and running.

2.2.2 Get dependencies

Run:

```
$ virtualenv ./venv/
$ . ./venv/bin/activate
$ pip install -r requirements-dev.txt
```

If you want to work on the contrib.django bits, you also need to do:

```
$ pip install -r requirements-django.txt
```

This sets up all the required dependencies for development of ElasticUtils.

Note: You don't have to put your virtual environment in `./venv/`. Feel free to put it anywhere.

2.3 Conventions

We follow the code conventions listed in the coding conventions page of the webdev bootcamp guide. This covers all the Python code.

We use git and follow the conventions listed in the git and github conventions page of the webdev bootcamp guide.

2.4 Documentation

2.4.1 Conventions

See the docmentation page in the webdev bootcamp guide for documentation conventions.

The documentation is available in HTML and PDF forms at http://elasticutils.readthedocs.org/. This tracks documentation in the master branch of the git repository. Because of this, it is always up to date.

2.4.2 Building the docs

The documentation in *docs/* is built with Sphinx. To build HTML version of the documentation, do:

```
$ cd docs/
$ make html
```

2.5 Running and writing tests

2.5.1 Running the tests

You can run the tests with:

```
./run_tests.py
```

If you have Django installed, this will run the Django-specific tests. If you don't, then the test runner will skip the Django-specific tests.

Note: If you need to adjust the settings, copy `test_settings.py` to a new file (like `test_settings_local.py`), edit the file, and specify that as the value for the environment variable `DJANGO_SETTINGS_MODULE`.

DJANGO_SETTINGS_MODULE=test_settings_local ./run_tests.py

This is helpful if you need to change the value of `ES_HOSTS` to match the ip address or port that elasticsearch is listening on.

2.5.2 Writing tests

Tests are located in *elasticutils/tests/*.

We use nose for test utilities and running tests.

2.5.3 ElasticTestCase

If you're testing things in ElasticUtils that require hitting an ElasticSearch instance, then you should subclass *elasticutils.tests.ElasticTestCase* which has code in it for making things easier.

class `elasticutils.tests.`**`ElasticTestCase`**(*methodName='runTest'*)

Superclass for ElasticSearch-using test cases.

Variables

- **index_name** – string; name of the index to use
- **skip_tests** – bool; if ElasticSearch isn't available, then this is True and therefore tests should be skipped for this class

For examples of usage, see the other `test_*.py` files.

classmethod `refresh`(*timesleep=0*)

Refresh index after indexing.

This refreshes the index specified by *self.index_name*.

Parameters **timesleep** – int; number of seconds to sleep after telling ES to refresh

`setUp`()

Set up a single test.

Raises SkipTest if `skip_tests` is True for this class/instance

classmethod `setup_class`()

Class setup for tests.

Checks to see if ES is running and if not, sets `skip_test` to True on the class.

classmethod `teardown_class`()

Class tear down for tests.

2.6 Release process

1. Checkout master tip.
2. Update version numbers in `elasticutils/_version.py`.
   (a) Set `__version__` to something like `0.4`.
   (b) Set `__releasedate__` to something like `20120731`.
3. Update `CONTRIBUTORS`, `CHANGELOG`, `MANIFEST.in`.
4. Verify correctness.
   (a) Run tests.
   (b) Build docs.
   (c) Verify all that works.
5. Tag the release:

   ```
   $ git tag -a v0.4
   ```

6. Push everything:

   ```
   $ git push --tags official master
   ```

7. Update PyPI:

   ```
   $ python setup.py sdist upload
   ```

8. Update topic in `#elasticutils`, blog post, twitter, etc.

CHAPTER

THREE

SAMPLE PROGRAMS

3.1 Sample with facets

```
"""
This is a sample program that uses PyES ES to create an index, create
a mapping, and index some data. Then it uses ElasticUtils S to show
some behavior with facets.
"""

from elasticutils import get_es, S


HOST = 'localhost:9200'
INDEX = 'fooindex'
DOCTYPE = 'testdoc'


es = get_es(hosts=HOST, default_indexes=[INDEX])

# This uses pyes ES.delete_index_if_exists to delete the index if it
# exists.
es.delete_index_if_exists(INDEX)

# Define the mapping for the doctype 'testdoc'. It's got an id field,
# a title which is analyzed, and two fields that are lists of tags, so
# we don't want to analyze them.
#
# Note: The alternative for the tags is to analyze them and use the
# 'keyword' analyzer. Both not analyzing and using the keyword
# analyzer treats the values as a single term rather than tokenizing
# them and treating as multiple terms.
mapping = {
    DOCTYPE: {
        'properties': {
            'id': {'type': 'integer'},
            'title': {'type': 'string'},
            'topics': {'type': 'string'},
            'product': {'type': 'string', 'index': 'not_analyzed'},
            }
        }
    }

# This uses pyes ES.create_index.
```

```
es.create_index(INDEX, settings={'mappings': mapping})


# This indexes a series of documents each is a Python dict.
for mem in [
    {'id': 1,
     'title': 'Deleting cookies',
     'topics': ['cookies', 'privacy'],
     'product': ['Firefox', 'Firefox for mobile']},
    {'id': 2,
     'title': 'What is a cookie?',
     'topics': ['cookies', 'privacy', 'basic'],
     'product': ['Firefox', 'Firefox for mobile']},
    {'id': 3,
     'title': 'Websites say cookies are blocked - Unblock them',
     'topics': ['cookies', 'privacy', 'websites'],
     'product': ['Firefox', 'Firefox for mobile', 'Boot2Gecko']},
    {'id': 4,
     'title': 'Awesome Bar',
     'topics': ['tips', 'search', 'basic', 'user interface'],
     'product': ['Firefox']},
    {'id': 5,
     'title': 'Flash',
     'topics': ['flash'],
     'product': ['Firefox']},]:

    es.index(mem, INDEX, DOCTYPE, id=mem['id'])

# ElasticSearch will refresh the indexes and make those documents
# available for querying in a second or so (it's configurable in
# ElasticSearch), but we want them available right now, so we refresh
# the index.
es.refresh(INDEX)

# Let's build a basic S that looks at the right instance of
# ElasticSearch, index, and doctype.
basic_s = (S().es(hosts=[HOST])
              .indexes(INDEX)
              .doctypes(DOCTYPE)
              .values_dict())

# Now let's see facet counts for all the products.
s = basic_s.facet('product')

print s.facet_counts()
# Pretty-printed output:
# {u'product': [
#    {u'count': 5, u'term': u'Firefox'},
#    {u'count': 3, u'term': u'Firefox for mobile'},
#    {u'count': 1, u'term': u'Boot2Gecko'}
#    ]}

# Let's do a query for 'cookie' and do a facet count.
print s.query(title__text='cookie').facet_counts()
# Pretty-printed output:
# {u'product': [
#    {u'count': 1, u'term': u'Firefox for mobile'},
#    {u'count': 1, u'term': u'Firefox'}
```

```
99  #     ]}
100
101 # Note that the facet_counts are affected by the query.
102
103 # Let's do a filter for 'flash' in the topic.
104 print s.filter(topics='flash').facet_counts()
105 # Pretty-printed output:
106 # {u'product': [
107 #     {u'count': 5, u'term': u'Firefox'},
108 #     {u'count': 3, u'term': u'Firefox for mobile'},
109 #     {u'count': 1, u'term': u'Boot2Gecko'}
110 #     ]}
111
112 # Note that the facet_counts are NOT affected by filters.
113
114 # Let's do a filter for 'flash' in the topic, and specify
115 # filtered=True.
116 print s.facet('product', filtered=True).filter(topics='flash').facet_counts()
117 # Pretty-printed output:
118 # {u'product': [
119 #     {u'count': 1, u'term': u'Firefox'}
120 #     ]}
121
122 # Using filtered=True causes the facet_counts to be affected by the
123 # filters.
124
125 # We've done a bunch of faceting on a field that is not
126 # analyzed. Let's look at what happens when we try to use facets on a
127 # field that is analyzed.
128 print basic_s.facet('topics').facet_counts()
129 # Pretty-printed output:
130 # {u'topics': [
131 #     {u'count': 3, u'term': u'privacy'},
132 #     {u'count': 3, u'term': u'cookies'},
133 #     {u'count': 2, u'term': u'basic'},
134 #     {u'count': 1, u'term': u'websites'},
135 #     {u'count': 1, u'term': u'user'},
136 #     {u'count': 1, u'term': u'tips'},
137 #     {u'count': 1, u'term': u'search'},
138 #     {u'count': 1, u'term': u'interface'},
139 #     {u'count': 1, u'term': u'flash'}
140 #     ]}
141
142 # Note how the facet counts shows 'user' and 'interface' as two
143 # separate terms even though they're a single topic for document with
144 # id=4. When that document is indexed, the topic field is analyzed and
145 # the default analyzer tokenizes it splitting it into two terms.
146 #
147 # Moral of the story is that you want fields you facet on to be
148 # analyzed as keyword fields or not analyzed at all.
```

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*

PYTHON MODULE INDEX

d

e