ElasticUtils Documentation

Release 0.7

Mozilla Foundation

June 12, 2013

CONTENTS

Version 0.7

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

ElasticUtils is a Python library that gives you a chainable search API for Elasticsearch as well as some other tools to make it easier to integrate Elasticsearch into your application.

So what's it like? Let's do a couple basic things:

Create an instance of `elasticutils.S` and tell it which index and doctype to look at.

```
>>> from elasticutils import S, F
>>> s = S().indexes('blog-index').doctypes('blog-entry')
```

Print the count of everything in that index with that type:

```
>>> s.count()
4
```

Show titles of all blog entries with "elasticutils" in the title:

```
>>> s = s.query(title__match='elasticutils')
>>> [result['title'] for result in s]
[u'ElasticUtils v0.4 released!', u'elasticutils status -- May 18th, 2012',
u'ElasticUtils sprint at PyCon US 2013']
```

You can also use properties rather than keys:

```
>>> [result.title for result in s]
[u'ElasticUtils v0.4 released!', u'elasticutils status -- May 18th, 2012',
u'ElasticUtils sprint at PyCon US 2013']
```

Filter out entries related to PyCon:

```
>>> s = s.filter(~F(tag='pycon'))
>>> [result['title'] for result in s]
[u'ElasticUtils v0.4 released!', u'elasticutils status -- May 18th, 2012']
```

Show only the top result:

```
>>> s = s[:1]
>>> [result['title'] for result in s]
[u'ElasticUtils v0.4 released!']
```

That's the gist of it!

CHAPTER

ONE

PROJECT

1.1 What's new in ElasticUtils

- Version 0.7: Released June 12th, 2013
- Version 0.6: Released January 17th, 2013
- Version 0.5: Released September 4th, 2012
- Version 0.4: Released July 31st, 2012
- Version 0.3: Released June 1st, 2012

1.1.1 Version 0.7: Released June 12th, 2013

Note: This is a *big* change. We switched from pyes to pyelasticsearch. In doing that, we changed a handful of signatures, nixed some functionality that didn't make any sense any more, and cleaned a bunch of things up.

If this terrifies you, read through these notes carefully and/or stay with v0.6.

API-breaking changes:

- **pyelasticsearch v0.4 or later now required.**

  ElasticUtils now requires pyelasticsearch v0.4 or later and its requirements.

- **elasticutils.PYES_VERSION is removed.**

  Since we're not using pyes, we removed *elasticutils.PYES_VERSION*.

- **ElasticUtils no longer supports thrift.**

  Pretty sure we did a lousy job of supporting it before—it was all in the pyes code and we had no tests for it.

- **get_es() signatures have changed.**

  - takes urls now instead of hosts
  - dump_curl argument is now gone
  - default_indexes argument is gone

  The arguments correspond with pyelasticsearch *ElasticSearch* object.

  ElasticUtils uses HTTP urls for connecting to Elasticsearch now. Previously, you'd do:

```
get_es(hosts=['localhost:9200'])         # Old way
```

Now you do:

```
get_es(urls=['http://localhost:9200'])   # New way
```

The dump_curl argument was helpful for debugging, but we don't really need it anymore. See the *Debugging* for better debugging methods.

Will now raise a *DeprecationWarning* if you pass in *hosts* argument.

- **S searches all indexes and doctypes by default.**

  Previously, if you did:

  ```
  S()
  ```

  it'd search an index named "default" for doctypes "document". That was dumb. Now it searches all indexes and all doctypes by default.

- **S.es_builder is gone.**

  `es_builder()` was there to get around problems with pyes' ES class. The pyelasticsearch *ElasticSearch* class is more straightforward, so we don't need to do circus shenanigans.

  You can probably do what you need to with either the `es()` transform or by subclassing *S* and overriding the `get_es()` method.

- **MLT arguments changed.**

  The *fields* argument in the constructor was renamed to *mlt_fields* to be in line with Elasticsearch API names.

  Will now raise a *DeprecationWarning* if you pass in *fields* argument.

- **MappingType get_indexes renamed to get_index.**

  MappingType had a method called *get_indexes*. This is now *get_index* because it should return a single index name.

- **Added Indexable mixin for indexing bits for MappingTypes.**

- **Django: changed settings.**

  Changed ES_HOSTS setting to ES_URLS. This is both a name and a value change. ES_URLS takes a list of strings each is an http url. You'll neex to update your settings files from:

  ```
  ES_HOSTS = ['localhost:9200']         # Old way
  ```

  to:

  ```
  ES_URLS = ['http://localhost:9200']   # New way
  ```

  ES_DUMP_CURL is gone.

- **Django: removed the statsd code.**

- **Django: ESTestCase was improved, documented and bugs squashed.**

  It was improved, documented and bugs were squashed. It's now used by the test suite.

- **Django: Indexable.index() method no longer has bulk argument.**

  The *Indexable.index()* method no longer does bulk indexing. The way pyes did this was kind of squirrely and caused issues if you didn't have the order of operations correct.

  Now *Indexable.index()* only indexes a single document.

But wait...

- **Django: Indexable now has bulk_index().**

  pyes would keep track of all the things you wanted to bulk index and then at some point push them all. Instead of doing it under the hood, we added a separate *bulk_index()* method and now you control how many items get indexed in bulk in one pass.

- **Django: Indexable.refresh_index no longer takes a timeout argument.**

  pyelasticsearch *ElasticSearch.refresh* doesn't take a timesleep argument, so we don't need that anymore.

- **Django: Indexable es argument defaults to Indexable.get_es() now.**

  Previously it defaulted to *elasticsearch.contrib.django.get_es()*. Now it defaults to *Indexable.get_es()* class method making it more flexible.

- **Django: renamed DjangoMappingType to MappingType.**

- **Django: moved MappingType and Indexable.**

  They were in `elasticutils.contrib.django.models` and are now in `elasticutils.contrib.django`. Yay for slightly shorter module paths!

- **Django: ditched the cron module and its helpers.**

  It's not clear they ever worked (issue #21) and there are no tests.

- **pyes -> pyelasticsearch changes.**

  If you called `.get_es()` and got a pyes *ES* object and did things with that (create index, create mappings, delete indexes, indexing, cluster health, ...), you're going to need to make some changes.

  You can either:

  1. rewrite that code to use pyelasticsearch *ElasticSearch* equivalents, or
  2. write and use your own `get_es()` function that returns a pyes *ES* object

  Rewriting shouldn't be too hard. The pyelasticsearch documentation is pretty good and for most things, there's a 1-to-1 translation.

Changes:

- **pyes is no longer a requirement.**

  We no longer use pyes so you can remove it from your requirements.

- **S.execute added**

  This allows you to explicitly execute a search and get back a *SearchResults* instance.

  See `elasticutils.S.execute()` for details.

- **S.all added**

  Allows you to get **all** the search results possible rather than just the first 10 search results which is the default.

  You should consider using slices instead which allows you to specify the maximum number of results to get back.

  This is dangerous, so it's been documented with lots of warnings.

  See `elasticutils.S.all()` for details.

- **Added support for ''match'' and ''match_phrase'' queries.**

  Elasticsearch 0.19.9 renamed text query to match query. This adds support for `match` and `match_phrase`.

See *Queries: query* for details.

- **Added support for ‘‘wildcard‘‘ and ‘‘terms‘‘ queries.**

  See *Queries: query* for details.

- **Reimplemented filter and query implementation.**

  The new implementations allow you to add handling for filters and queries that ElasticUtils doesn't handle as well as override what ElasticUtils does.

  See `elasticutils.S` for details.

- **S.query_raw added**

  If `elasticutils.S.query()` is getting you down, then you can skip it and use the Elasticsearch API to create the query clause of the search by hand with `elasticutils.S.query_raw()`.

- **Django: es_required_or_50x handles different exceptions.**

  Previously it handled:

  - pyes.urllib3.MaxRetryError
  - pyes.exceptions.IndexMissingException
  - pyes.exceptions.ElasticSearchException

  We're not using pyes anymore, so now it handles:

  - pyelasticsearch.exceptions.ConnectionError
  - pyelasticsearch.exceptions.ElasticHttpError
  - pyelasticsearch.exceptions.ElasticHttpNotFoundError
  - pyelasticsearch.exceptions.InvalidJsonResponseError
  - pyelasticsearch.exceptions.Timeout

  You probably don't need to do anything about this, but it's good to know.

- **Django: celery tasks rewritten.**

  The celery tasks were rewritten, docs were updated, and tests were added so they work now.

1.1.2 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')
  ```

1.1.3 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.

1.1.4 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.

1.1.5 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 Elasticsearch theory

1.2.1 Indexes and types

Elasticsearch stores documents in an index allowing you to search them. The index is a container for documents. You can have multiple indexes in your cluster of Elasticsearch nodes.

Documents are typed. A type has a list of fields that are in the documents of that type. ElasticUtils calls this a "mapping type" or a "doc type" since the word "type" is somewhat ambiguous depending on the context.

See Also:

http://www.elasticsearch.org/guide/reference/glossary/#index Elasticsearch explanation of indexes

http://www.elasticsearch.org/guide/reference/glossary/#mapping Elasticsearch explanation of mappings

http://www.elasticsearch.org/guide/reference/glossary/#type Elasticsearch explanation of types

1.2.2 Queries vs. filters

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

A **filter** determines whether a document is in the results set or not. It doesn't affect scores. 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. Use filters when you can.

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 Resources

1.3.1 Documentation

Elasticsearch guide

> http://www.elasticsearch.org/guide/
>
> This is the canonical documentation.

pyelasticsearch documentation

> https://pyelasticsearch.readthedocs.org/en/latest/
>
> ElasticUtils sits on top of pyelasticsearch, so their documentation is very helpful.

1.3.2 Videos

Elasticsearch videos

> http://www.elasticsearch.org/videos/
>
> Lots of videos covering a variety of use cases and other things.

Elasticsearch video tutorials

> http://www.elasticsearch.org/tutorials/
>
> Covers deployment and using Elasticsearch for various things

FoodFightShow

> http://www.youtube.com/watch?v=dBWlXdmjjzY
>
> Covers Elasticsearch.

Erik Rose's talks:

> http://pyvideo.org/video/1784/elasticsearch-part-1-indexing-and-querying
>
> Elasticsearch provides an easy path to clusterable full-text search, with synonyms, faceting, and geographic math, but there's a paucity of written wisdom beyond its API docs. This talk, part 1 of a 2-part series, surveys its capabilities and shows how its internal data structures and algorithms work. With the

groundwork laid, we explore how to choose efficient indexing and the right queries to make your apps go fast.

CHAPTER

TWO

USER'S GUIDE

2.1 Installation

Warning: ElasticUtils doesn't work well with Elasticsearch 0.19.9. Don't use that version—newer or older versions are fine.
https://github.com/elasticsearch/elasticsearch/issues/2205 Elasticsearch bug with `_all`.

2.1.1 Requirements

ElasticUtils requires:

- pyelasticsearch >= 0.4

2.1.2 Installation

There are a few ways to install ElasticUtils:

From PyPI

Do:

```
$ pip install elasticutils
```

From git

Do:

```
$ git clone git://github.com/mozilla/elasticutils.git
$ cd elasticutils
$ python setup.py install
```

2.2 Indexing

- Overview
- Getting an ElasticSearch object
- Indexes
- Types and Mappings
- Indexing documents
- Deleting documents
- Refreshing
- Delete indexes
- Doing all of this with MappingTypes and Indexables

2.2.1 Overview

ElasticUtils is primarily an API for searching. However, before you can search, you need to create an index and index your documents.

This chapter covers the indexing side of things. It does so lightly—for more details, read through the pyelasticsearch documentation and the Elasticsearch guide.

2.2.2 Getting an ElasticSearch object

ElasticUtils uses *pyelasticsearch* which comes with a handy *ElasticSearch* object. This lets you:

- create indexes
- create mappings
- apply settings
- check status
- etc.

To access this, you use `elasticutils.get_es()` which creates an *ElasticSearch* object for you.

See `elasticutils.get_es()` for more details.

See Also:

http://pyelasticsearch.readthedocs.org/en/latest/api/ pyelasticsearch ElasticSearch documentation.

2.2.3 Indexes

An *index* is a collection of documents.

Before you do anything, you need to have an index. You can create one with *.create_index()*.

For example:

```
es = get_es()
es.create_index('blog-index')
```

You can pass in settings, too. For example, you can set the refresh interval when creating the index:

```
es.create_index('blog-index', settings={'refresh_interval': '5s'})
```

See Also:

http://pyelasticsearch.readthedocs.org/en/latest/api/#pyelasticsearch.ElasticSearch.create_index pyelasticsearch create_index API documentation

http://www.elasticsearch.org/guide/reference/api/admin-indices-create-index/ Elasticsearch create_index API documentation

2.2.4 Types and Mappings

A *type* is a set of fields. A document is of a given type if it has those fields. Whenever you index a document, you specify which type the document is. This is sometimes called a "doctype", "document type" or "doc type".

A *mapping* is the definition of fields and how they should be indexed for a type. In ElasticUtils, we call a document type that has a defined mapping a "mapping type" mostly as a shorthand for "document type with a defined mapping" because that's a mouthful.

Elasticsearch can infer mappings to some degree, but you get a lot more value by specifying mappings explicitly.

To define a mapping, you use *.put_mapping()*.

For example:

```
es = get_es()
es.put_mapping('blog-index', 'blog-entry-type', {
    'id': {'type': 'integer'},
    'title': {'type': 'string'},
    'content': {'type': 'string'},
    'tags': {'type': 'string'},
    'created': {'type': 'date'}
    })
```

You can also define mappings when you create the index:

```
es = get_es()
es.create_index('blog-index', settings={
    'mappings': {
        'blog-entry-type': {
            'id': {'type': 'integer'},
            'title': {'type': 'string'},
            'content': {'type': 'string'},
            'tags': {'type': 'string'},
            'created': {'type': 'date'}
        }}})
```

Note: If there's a possibility of a race condition between creating the index and defining the mapping and some document getting indexed, then it's good to create the index and define the mappings at the same time.

See Also:

http://pyelasticsearch.readthedocs.org/en/latest/api/#pyelasticsearch.ElasticSearch.put_mapping pyelasticsearch put_mapping API documentation

http://www.elasticsearch.org/guide/reference/api/admin-indices-put-mapping/ Elasticsearch put_mapping API documentation

http://www.elasticsearch.org/guide/reference/mapping/ Elasticsearch mapping documentation

2.2.5 Indexing documents

Use *.index()* to index a document.

For example:

```
es = get_es()

entry = {'id': 1,
    'title': 'First post!',
    'content': '<p>First post!</p>',
    'tags': ['status', 'blog'],
    'created': '20130423T16:50:22'
    }

es.index('blog-index', 'blog-entry-type', entry, 1)
```

If you're indexing a bunch of documents at the same time, you should use *.bulk_index()*.

For example:

```
es = get_es()

entries = { ... }

es.bulk_index('blog-index', 'blog-entry-type', entries, id_field='id')
```

See Also:

http://pyelasticsearch.readthedocs.org/en/latest/api/#pyelasticsearch.ElasticSearch.index pyelasticsearch index API documentation

http://pyelasticsearch.readthedocs.org/en/latest/api/#pyelasticsearch.ElasticSearch.bulk_index pyelasticsearch bulk_index API documentation

http://www.elasticsearch.org/guide/reference/api/index_/ Elasticsearch index API documentation

http://www.elasticsearch.org/guide/reference/api/bulk/ Elasticsearch bulk index API documentation

2.2.6 Deleting documents

You can delete documents with *.delete()*.

For example:

```
es = get_es()

es.delete('blog-index', 'blog-entry-type', 1)
```

See Also:

http://pyelasticsearch.readthedocs.org/en/latest/api/#pyelasticsearch.ElasticSearch.delete pyelasticsearch delete API documentation

http://www.elasticsearch.org/guide/reference/api/delete/ Elasticsearch delete API documentation

2.2.7 Refreshing

After you index documents, they're not available for searches until after the index is refreshed. By default, the index refreshes every second. If you need the documents to show up in searches before that, call *.refresh()*.

For example:

```
es = get_es()

es.refresh('blog-index')
```

See Also:

http://pyelasticsearch.readthedocs.org/en/latest/api/#pyelasticsearch.ElasticSearch.refresh pyelasticsearch refresh API documentation

http://www.elasticsearch.org/guide/reference/api/admin-indices-refresh/ Elasticsearch refresh API documentation

2.2.8 Delete indexes

You can delete indexes with *.delete_index()*.

For example:

```
es = get_es()

es.delete_index('blog-index')
```

See Also:

http://pyelasticsearch.readthedocs.org/en/latest/api/#pyelasticsearch.ElasticSearch.delete_index pyelasticsearch delete_index API documentation

http://www.elasticsearch.org/guide/reference/api/admin-indices-delete-index/ Elasticsearch delete index API documentation

2.2.9 Doing all of this with MappingTypes and Indexables

If you're using MappingTypes, then you can do much of the above using methods and classmethods on `MappingType` and `Indexable` classes. See *Mapping types and Indexables* for more details.

2.3 Mapping types and Indexables

2.3.1 The MappingType class

`elasticutils.MappingType` lets you centralize concerns regarding documents you're storing in your Elasticsearch index.

Lets you tie business logic to search results

When you do searches with MappingTypes, you get back those results as an iterable of MappingTypes by default.

For example, say you had a description field and wanted to have a truncated version of it. You could do it this way:

```
class MyMappingType(MappingType):

    # ... missing code here

    def description_truncated(self):
```

```
        return self.description[:100]

results = S(MyMappingType).query(description__text='stormy night')

print list(results)[0].description_truncated()
```

Lets you link source data to search results

You can relate a MappingType to a database model or other source allowing you to link documents in the Elasticsearch index back to their origins in a lazy-loading way. This is done by subclassing MappingType and implementing the `get_object()` method. You can then access the original data using the `object` property.

For example:

```
class MyMappingType(MappingType):

    # ... missing code here

    def get_object(self):
        return self.get_model().objects.get(pk=self._id)

results = S(MyMappingType).filter(height__gte=72)[:1]

first = list(results)[0]

# This prints "height" which comes from the Elasticsearch
# document
print first.height

# This prints "height" which comes from the database data
# that the Elasticsearch document is based on. This is the
# first time ``.object`` is used, so it does the db hit
# here.
print first.object.height
```

DefaultMappingType

The most basic MappingType is the DefaultMappingType which is returned if you don't specify a MappingType and also don't call `elasticutils.S.values_dict()` or s:py:meth:*elasticutils.S.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 or aren't valid Python names.

For more information

See *Types and Mappings* for documentation on defining mappings in the index.

See `elasticutils.MappingType` for documentation on creating MappingTypes.

2.3.2 The Indexable class

`elasticutils.Indexable` is a mixin for `elasticutils.MappingType` that has methods and classmethods for making indexing easier.

2.3.3 Example

Here's an example of a class that subclasses *MappingType* and *Indexable*. It's based on a model called *BlogEntry*.

```
class BlogEntryMappingType(MappingType, Indexable):
    @classmethod
    def get_index(cls):
        return 'blog-index'

    @classmethod
    def get_mapping_type_name(cls):
        return 'blog-entry'

    @classmethod
    def get_model(cls):
        return BlogEntry

    @classmethod
    def get_es(cls):
        return get_es(urls=['http://localhost:9200'])

    @classmethod
    def get_mapping(cls):
        return {
            'properties': {
                'id': {'type': 'integer'},
                'title': {'type': 'string'},
                'tags': {'type': 'string'}
            }
        }

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

        doc = {}
        doc['id'] = obj.id
        doc['title'] = obj.title
        doc['tags'] = obj.tags
        return doc

    @classmethod
    def get_indexable(cls):
        return cls.get_model().get_objects()
```

With this, I can write code elsewhere in my project that:

1. gets the mapping type name and mapping for documents of type "blog-entry"
2. gets all the objects that are indexable
3. for each object, extracts the Elasticsearch document data and indexes it

When I create my `elasticutils.S` object, I'd create it like this:

```
s = S(BlogEntryMappingType)
```

and now by default any search results I get back are instances of the *BlogEntryMappingType* class.

2.4 Searching

- Overview
- All about S: `S`
  - What is S?
  - S is chainable
  - S can be typed and untyped
  - S can be sliced to return the results you want
  - S is lazy
  - S results can be returned in many shapes
- Where to search
  - Specifying connection parameters: `es`
  - Specifying indexes to search: `indexes`
  - Specifying doctypes to search: `doctypes`
- By default, S does a Match All
- Queries: `query`
- Advanced queries: `Q` and `query_raw`
  - calling .query() multiple times
  - should, must and must_not
  - The Q class
  - query_raw
  - adding new query actions
- Filters: `filter`
- Advanced filters: `F`
  - and vs. or
  - The F class
  - adding new filteractions
- Query-time field boosting: `boost`
- Ordering: `order_by`
- Demoting: `demote`
- Highlighting: `highlight`
- Facets
  - Basic facets: `facet`
  - Facets and scope (filters and global)
  - Facets... RAW!: `facet_raw`
- Scores and explanations
  - Seeing the score: _score
  - Getting an explanation: `explain`

2.4.1 Overview

This chapter covers how to search with ElasticUtils.

2.4.2 All about S: S

What is S?

elasticutils.S helps you define an Elasticsearch search.

```
searcher = S()
```

This creates an *untyped* elasticutils.S using the defaults:

- uses an pyelasticsearch.client.ElasticSearch instance configured to connect to http://localhost:9200 – call elasticutils.S.es() to specify connection parameters
- searches across all indexes – call elasticutils.S.indexes() to specify indexes
- searches across all doctypes – call elasticutils.S.doctypes() to specify doctypes

S is chainable

elasticutils.S has methods that return a new S instance with the additional specified criteria. In this way S is chainable and you can reuse S objects for your searches.

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* with a awesome=True filter.

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

S can be typed and untyped

When you create an elasticutils.S with no type, it's called an *untyped S*. By default, search results for a *untyped S* are returned in the form of a sequence of elasticutils.DefaultMappingType instances. You can explicitly state that you want a sequence of dicts or lists, too. See *S results can be returned in many shapes* for more details on how to return results in various formats.

You can also construct a *typed S* which is an *S* with a elasticutils.MappingType subclass. By default, search results for a *typed S* are returned in the form of a sequence of instances of that type. See *Mapping types and Indexables* for more about MappingTypes.

S can be sliced to return the results you want

By default Elasticsearch gives you the first 10 results.

If you want something different than that, elasticutils.S supports slicing allowing you to get back the specific results you're looking for.

For example:

```
some_s = S()

results = some_s[:10]     # returns first 10 results
results = some_s[10:20]   # returns results 10 through 19
```

The slicing is chainable, too:

```
some_s = S()[:10]
first_ten_pitchers = some_s.filter(position='pitcher')
```

Note: The slicing happens on the Elasticsearch side—it doesn't pull all the results back and then slice them in Python. Ew.

See Also:

http://www.elasticsearch.org/guide/reference/api/search/from-size.html Elasticsearch from / size documentation

S is lazy

The search won't execute until you do one of the following:

1. use the `elasticutils.S` in an iterable context
2. call `len()` on a `elasticutils.S`
3. call the `elasticutils.S.execute()`, `elasticutils.S.all()`, `elasticutils.S.count()` or `elasticutils.S.facet_counts()` methods

Once you execute the search, then it will cache the results and further executions of that `elasticutils.S` won't result in another roundtrip to your Elasticsearch cluster.

S results can be returned in many shapes

An *untyped S* (e.g. `S()`) will return instances of `elasticutils.DefaultMappingType` by default.

A *typed S* (e.g. `S(Foo)`), will return instances of that type (e.g. type `Foo`) by default.

`elasticutils.S.values_list()` gives you a list of tuples. See documentation for more details.

`elasticutils.S.values_dict()` gives you a list of dicts. See documentation for more details.

If you use `elasticutils.S.execute()`, you get back a `elasticutils.SearchResults` instance which has additional useful bits including the raw response from Elasticsearch. See documentation for details.

2.4.3 Where to search

Specifying connection parameters: `es`

`elasticutils.S` will generate an `pyelasticsearch.client.ElasticSearch` object that connects to `http://localhost:9200` by default. That's usually not what you want. You can use the `elasticutils.S.es()` method to specify the arguments used to create the pyelasticsearch ElasticSearch object.

Examples:

```
q = S().es(urls=['http://localhost:9200'])
q = S().es(urls=['http://localhost:9200'], timeout=10)
```

See `elasticutils.get_es()` for the list of arguments you can pass in.

Specifying indexes to search: `indexes`

An *untyped S* will search all indexes by default.

A *typed S* will search the index returned by the `elasticutils.MappingType.get_index()` method.

If that's not what you want, use the `elasticutils.S.indexes()` method.

For example, this searches all indexes:

```
q = S()
```

This searches just "someindex":

```
q = S().indexes('someindex')
```

This searches "thisindex" and "thatindex":

```
q = S().indexes('thisindex', 'thatindex')
```

Specifying doctypes to search: `doctypes`

An *untyped S* will search all doctypes by default.

A *typed S* will search the doctype returned by the `elasticutils.MappingType.get_mapping_type_name()` method.

If that's not what you want, then you should use the `elasticutils.S.doctypes()` method.

For example, this searches all doctypes:

```
q = S()
```

This searches just the "sometype" doctype:

```
q = S().doctypes('sometype')
```

This searches "thistype" and "thattype":

```
q = S().doctypes('thistype', 'thattype')
```

2.4.4 By default, S does a Match All

By default, `elasticutils.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

2.4.5 Queries: `query`

Queries are specified using the `elasticutils.S.query()` method. See those docs for API details.

ElasticUtils uses this syntax for specifying queries:

> fieldname__fieldaction=value

1. fieldname: the field the query applies to
2. fieldaction: the kind of query it is
3. value: the value to query for

The fieldname and fieldaction are separated by __ (that's two underscores).

For example:

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

will do an Elasticsearch match query on the title field for "taco trucks".

There are many different field actions to choose from:

field action	elasticsearch query type
(no action specified)	Term query
term	Term query
terms	Terms query
text	Text query
match	Match query [1]
prefix	Prefix query [2]
gt, gte, lt, lte	Range query
fuzzy	Fuzzy query
wildcard	Wildcard query
text_phrase	Text phrase query
match_phrase	Match phrase query [1]
query_string	Querystring query [3]

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/terms-query.html Elasticsearch docs on terms 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/match-query.html Elasticsearch docs on match and match_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

[1] Elasticsearch 0.19.9 renamed text queries to match queries. If you're using Elasticsearch 0.19.9 or later, you should use match and match_phrase. If you're using a version prior to 0.19.9 use text and text_phrase.

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

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

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

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

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

2.4.6 Advanced queries: `Q` and `query_raw`

calling .query() multiple times

Calling `elasticutils.S.query()` multiple times will combine all the queries together.

should, must and must_not

By default all queries must match a document in order for the document to show up in the search results.

You can alter this behavior by flagging your queries with `should`, `must`, and `must_not` flags.

should

A query added with `should=True` affects the score for a result, but it won't prevent the document from being in the result set.

Example:

```
qs = S().query(title__text='castle',
               summary__text='castle',
               should=True)
```

If the document matches either the `title__text` or the `summary__text` then it's included in the results set. It doesn't *have* to match both.

must

This is the default.

A query added with `must=True` must match in order for the document to be in the result set.

Example:

```
qs = S().query(title__text='castle',
               summary__text='castle')

qs = S().query(title__text='castle',
               summary__text='castle',
               must=True)
```

These two are equivalent. The document must match both the `title__text` and `summary__text` queries in order to be included in the result set. If it doesn't match one of them, then it's not included.

must_not

A query added with `must_not=True` must NOT match in order for the document to be in the result set.

Example:

```
qs = (S().query(title__text='castle')
         .query(author='castle', must_not=True))
```

For a document to be included in the result set, it must match the `title__text` query and must NOT match the `author` query. I.e. The title must have "castle", but the document can't have been written by someone with "castle" in their name.

The Q class

You can manipulate query units with the `elasticutils.Q` class. For example, you can incrementally build your query:

```
q = Q()

if search_authors:
    q += Q(author_name=search_text, should=True)

if search_keywords:
    q += Q(keyword=search_text, should=True)

q += Q(title__text=search_text, summary__text=search_text,
       should=True)
```

The + Python operator will combine two *Q* instances together and return a new instance.

You can then use one or more *Q* classes in a query call:

```
if search_authors:
    q += Q(author_name=search_text, should=True)

if search_keywords:
    q += Q(keyword=search_text, should=True)

q += Q(title__text=search_text, summary__text=search_text,
       should=True)

s = S().query(q)
```

query_raw

`elasticutils.S.query_raw()` lets you explicitly define the query portion of an Elasticsearch search.

For example:

```
q = S().query_raw({'match': {'title': 'example'}})
```

This will override all `.query()` calls you've made in your `elasticutils.S` before and after the *.query_raw* call.

This is helpful if ElasticUtils is missing functionality you need.

adding new query actions

You can subclass `elasticutils.S` and add handling for additional query actions. This is helpful in two circumstances:

1. ElasticUtils doesn't have support for that query type
2. ElasticUtils doesn't support that query type in a way you need—for example, ElasticUtils uses different argument values

See `elasticutils.S` for more details on how to do this.

2.4.7 Filters: `filter`

Filters are specified using the `elasticutils.S.filter()` method. See those docs for API details.

```
q = S().filter(language='korean')
```

will do a search and only return results where the language is Korean.

`elasticutils.S.filter()` uses the same syntax for specifying fields, actions and values as `elasticutils.S.query()`.

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

You can also filter on fields that have `None` as a value or have no value:

```
q = S().filter(language=None)
```

This uses the Elasticsearch Missing filter.

Note: In order to filter on fields that have `None` as a value, you have to tell Elasticsearch that the field can have null values. To do this, you have to add `null_value: True` to the mapping for that field.

http://www.elasticsearch.org/guide/reference/mapping/core-types.html

---

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

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

2.4.8 Advanced filters: `F`

and vs. or

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.

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

For example:

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

The F class

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'})
```

But, wow—that's icky looking and not particularly helpful!

So, we've also got an `elasticutils.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.

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'))
```

F supports `&` (and), `|` (or) and ~ (not) operations.

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.

adding new filteractions

You can subclass `elasticutils.S` and add handling for additional filter actions. This is helpful in two circumstances:

1. ElasticUtils doesn't have support for that filter type
2. ElasticUtils doesn't support that filter type in a way you need—for example, ElasticUtils uses different argument values

See `elasticutils.S` for more details on how to do this.

2.4.9 Query-time field boosting: `boost`

ElasticUtils allows you to specify query-time field boosts with `elasticutils.S.boost()`.

These boosts take effect at the time the query is executing. After the query has executed, then the boost is applied and that becomes the final score for the query.

This is a useful way to weight queries for some fields over others.

See elasticutils.S.boost() for more details.

Note: Boosts are ignored if you use query_raw.

2.4.10 Ordering: order_by

ElasticUtils elasticutils.S.order_by() lets you change the order of the search results.

See elasticutils.S.order_by() for more details.

See Also:

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

2.4.11 Demoting: demote

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 elasticutils.S.demote() once. Calling it again overwrites previous calls.

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

2.4.12 Highlighting: highlight

ElasticUtils can highlight excerpts for search results.

See elasticutils.S.highlight() for more details.

See Also:

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

2.4.13 Facets

Basic facets: `facet`

```
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.

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

The facet counts are available through `elasticutils.S.facet_counts()`. 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'))
```

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.

For example:

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

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

For example:

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

For example:

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

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!: `facet_raw`

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

You could do the first facet example with:

```
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'
        }))
```

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

See Also:

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

2.4.14 Scores and explanations

Seeing the score: _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: `explain`

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 `elasticutils.S.explain()`.

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).

2.5 More like this: `MLT`

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 with id 2034 of type *addon* in the *addon_index*.

You can pass it an *S* instance and the *MLT* will derive the index, doctype, ElasticSearch 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)
```

See `elasticutils.MLT` for more details.

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.

http://pyelasticsearch.readthedocs.org/en/latest/api/#pyelasticsearch.ElasticSearch.more_like_this pyelasticsearch documentation for MLT

2.6 Debugging

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

2.6.1 Score explanations

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

2.6.2 Logging

pyelasticsearch logs to the `pyelasticsearch` logger using the Python logging module. If you configure that to show DEBUG-level messages, then it'll show the requests in curl form, responses, and when it marks servers as dead.

Additionally, pyelasticsearch uses Requests which logs to the `requests` logger using the Python logging module. If you configure that to show INFO-level messages, then you'll see all that stuff.

First set up logging using something like this:

```
import logging

# Set up the logging in some way. If you don't have logging
# set up, you can set it up like this.
logging.basicConfig()
```

Then set the logging level for the pyelasticsearch and requests loggers to `logging.DEBUG`:

```
logging.getLogger('pyelasticsearch').setLevel(logging.DEBUG)
logging.getLogger('requests').setLevel(logging.DEBUG)
```

pyelasticsearch will log lines like:

```
DEBUG:pyelasticsearch:Making a request equivalent to this: curl
-XGET 'http://localhost:9200/fooindex/testdoc/_search' -d '{"fa
cets": {"topics": {"terms": {"field": "topics"}}}}'
```

You can copy and paste the curl line and it'll work on the command line.

Note: If you add a `pretty=1` to the query string of the url that you're curling, then Elasticsearch will return a prettified response that's easier to read.

2.6.3 Seeing the query

The *S* class has a *_build_query()* method that you can use to see the body of the Elasticsearch request it's generated with the parameters you've specified so far. This is helpful in debugging ElasticUtils and figuring out whether it's doing things poorly.

For example:

```
some_s = S()
print some_s._build_query()
```

Note: This is a "private" method, so we might change it at some point. Having said that, it hasn't changed so far and it is super helpful.

2.6.4 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.

2.6.5 elasticsearch-paramedic

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

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

2.6.6 es2unix

https://github.com/elasticsearch/es2unix

Use this for calling Elasticsearch API things instead of curl.

2.7 API docs

- Functions
- The S class
- The F class
- The Q class
- The SearchResults class
- The MappingType class
- The Indexable class
- The DefaultMappingType class
- The MLT class

2.7.1 Functions

`elasticutils.`**`get_es`**(*urls=None*, *timeout=5*, *force_new=False*, ***settings*)

Create a pyelasticsearch *ElasticSearch* object and return it.

This will aggressively re-use *ElasticSearch* objects with the following rules:

1. if you pass the same argument values to *get_es()*, then it will return the same *ElasticSearch* object
2. if you pass different argument values to *get_es()*, then it will return different *ElasticSearch* object
3. it caches each *ElasticSearch* object that gets created
4. if you pass in *force_new=True*, then you are guaranteed to get a fresh *ElasticSearch* object AND that object will not be cached

Parameters

- **urls** – list of uris; Elasticsearch hosts to connect to, defaults to `['http://localhost:9200']`
- **timeout** – int; the timeout in seconds, defaults to 5
- **force_new** – Forces get_es() to generate a new ElasticSearch object rather than pulling it from cache.

- **settings** – other settings to pass into ElasticSearch constructor; See http://pyelasticsearch.readthedocs.org/en/latest/api/ for more details.

Examples:

```
# Returns cached ElasticSearch object
es = get_es()

# Returns a new ElasticSearch object
es = get_es(force_new=True)

es = get_es(urls=['http://localhost:9200'])

es = get_es(urls=['http://localhost:9200'], timeout=10,
            max_retries=3)
```

2.7.2 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`.

Adding support for other queries

You can add support for queries that S doesn't have support for by subclassing S with a method called `process_query_ACTION`. This method takes a key, value and an action.

For example:

```
claass FunkyS(S):
    def process_query_funkyquery(self, key, val, action):
        return {'funkyquery': {'field': key, 'value': val}}
```

Then you can use that just like other actions:

```
s = FunkyS().query(Q(foo__funkyquery='bar'))
s = FunkyS().query(foo__funkyquery='bar')
```

Many Elasticsearch queries take other arguments. This is a good way of using different arguments. For example, if you wanted to write a handler for fuzzy for dates, you could do:

```
claass FunkyS(S):
    def process_query_fuzzy(self, key, val, action):
        # val here is a (value, min_similarity) tuple
        return {
            'funkyquery': {
                key: {
                    'value': val[0],
                    'min_similarity': val[1]
                }
            }
        }
```

Used:

```
s = FunkyS().query(created__fuzzy=(created_dte, '1d'))
```

Adding support for other filters

You can add support for filters that S doesn't have support for by subclassing S with a method called `process_filter_ACTION`. This method takes a key, value and an action.

For example:

```
claass FunkyS(S):
    def process_filter_funkyfilter(self, key, val, action):
        return {'funkyfilter': {'field': key, 'value': val}}
```

Then you can use that just like other actions:

```
s = FunkyS().filter(F(foo__funkyfilter='bar'))
s = FunkyS().filter(foo__funkyfilter='bar')
```

__init__(*type_=None*)

Create and return an S.

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

Chaining transforms

query(**queries*, ***kw*)

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

Parameters

- **queries** – instances of Q
- **kw** – queries in the form of `field__action=value`

There are three special flags you can use:

- `must=True`: Specifies that the queries and kw queries **must match** in order for a document to be in the result.

  If you don't specify a special flag, this is the default.
- `should=True`: Specifies that the queries and kw queries **should match** in order for a document to be in the result.
- `must_not=True`: Specifies the queries and kw queries **must not match** in order for a document to be in the result.

These flags work by putting those queries in the appropriate clause of an Elasticsearch boolean query.

Examples:

```
>>> s = S().query(foo='bar')
>>> s = S().query(Q(foo='bar'))
>>> s = S().query(foo='bar', bat__text='baz')
>>> s = S().query(foo='bar', should=True)
>>> s = S().query(foo='bar', should=True).query(baz='bat', must=True)
```

Notes:

1. Don't specify multiple special flags, but if you did, *should* takes precedence.
2. If you don't specify any, it defaults to *must*.
3. You can specify special flags in the `elasticutils.Q`, too. If you're building your query incrementally, using `elasticutils.Q` helps a lot.

See the documentation on `elasticutils.Q` for more details on composing queries with Q.

See the documentation on `elasticutils.S` for more details on adding support for more query types.

query_raw(*query*)

Return a new S instance with a query_raw.

Parameters **query** – Python dict specifying the complete query to send to Elasticsearch

Example:

```
S().query_raw({'match': {'title': 'example'}})
```

Note: If there's a query_raw in your S, then that's your query. All `.query()`, `.demote()`, `.boost()` and anything else that affects the query clause is ignored.

---

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

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

Parameters

- **filters** – this will be instances of F
- **kw** – this will be in the form of `field__action=value`

Examples:

```
>>> s = S().filter(foo='bar')
>>> s = S().filter(F(foo='bar'))
>>> s = S().filter(foo='bar', bat='baz')
>>> s = S().filter(foo='bar').filter(bat='baz')
```

By default, everything is combined using AND. If you provide multiple filters in a single filter call, those are ANDed together. If you provide multiple filters in multiple filter calls, those are ANDed together.

If you want something different, use the F class which supports `&` (and), `|` (or) and `~` (not) operators. Then call filter once with the resulting F instance.

See the documentation on `elasticutils.F` for more details on composing filters with F.

See the documentation on `elasticutils.S` for more details on adding support for new filter types.

order_by(**fields*)

Return a new S instance with results ordered as specified

You can change the 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`.

Note: Calling this again will overwrite previous `.order_by()` calls.

---

boost(**kw)

Return a new S instance with field boosts.

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.

Examples:

```
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.

Boosts are relative to one another and all boosts default to 1.0.

For example, if you had:

```
qs = (S().boost(title=4.0, summary=2.0)
         .query(title__text=value,
                summary__text=value,
                content__text=value,
                should=True))
```

`title__text` would be boosted twice as much as `summary__text` and `summary__text` twice as much as `content__text`.

demote(*amount_*, **queries*, ***kw*)

Returns a new S instance with boosting query and demotion.

You can demote documents that match query criteria:

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

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

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: Calling this again will overwrite previous `.demote()` calls.

---

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.

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

Results will have a `_highlight` property which contains the highlighted field excerpts.

For example:

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

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

If you pass in `None`, it will clear the highlight.

For example, this search won't highlight anything:

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

Note: Calling this again will overwrite previous `.highlight()` calls.

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

---

values_list(**fields*)
Return a new S instance that returns ListSearchResults.

Parameters **fields** – the list of fields to have in the results.

With no arguments, returns a list of tuples of all the data for that document.

With arguments, returns a list of tuples where the fields in the tuple are in the order 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`.

---

values_dict(**fields*)

Return a new S instance that returns DictSearchResults.

Parameters **fields** – the list of fields to have in the results.

With no arguments, this returns a list of dicts with all the fields.

With arguments, it returns a list of dicts with the specified fields.

For example:

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

es(***settings*)

Return a new S with specified ElasticSearch settings.

This allows you to configure the ElasticSearch object that gets used to execute the search.

Parameters **settings** – the settings you'd use to build the ElasticSearch—same as what you'd pass to `get_es()`.

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 0x2a76938>*)

Returns the ElasticSearch object to use.

Parameters **default_builder** – The function that takes a bunch of arguments and generates a pyelasticsearch ElasticSearch object.

Note: If you desire special behavior regarding building the ElasticSearch object for this S, subclass S and override this method.

---

get_indexes(*default_indexes=None*)

Returns the list of indexes to act on.

get_doctypes(*default_doctypes=None*)

Returns the list of doctypes to use.

to_python(*obj*)

Converts strings in a data structure to Python types

It converts datetime-ish things to Python datetimes.

Override if you want something different.

Parameters **obj** – Python datastructure

Returns Python datastructure with strings converted to Python types

---

Note: This does the conversion in-place!

Methods that force evaluation

__iter__()

Executes search and returns an iterator of results.

Returns iterator of results

For example:

```
>>> s = S().query(name__prefix='Jimmy')
>>> for obj in s.execute():
...     print obj['id']
...
```

__len__()

Executes search and returns the number of results you'd get.

Executes search and returns number of results as an integer.

Returns integer

For example:

```
>>> s = S().query(name__prefix='Jimmy')
>>> count = len(s)
>>> results = s().execute()
>>> count = len(results)
True
```

Note: This is very different than calling `.count()`. If you call `.count()` you get the total number of results that Elasticsearch thinks matches your search. If you call `len(s)`, then you get the number of results you'd get if you executed the search. This factors in slices and default from and size values.

---

all()

Executes search and returns ALL search results.

Returns *SearchResults* instance

For example:

```
>>> s = S().query(name__prefix='Jimmy')
>>> all_results = s.all()
```

Warning: This returns ALL search results. The way it does this is by calling `.count()` first to figure out how many to return, then by slicing by that size and returning a list of ALL search results.
Don't use this if you've got 1000s of results!

count()

Executes search and returns number of results as an integer.

Returns integer

For example:

```
>>> s = S().query(name__prefix='Jimmy')
>>> count = s.count()
```

execute()
Executes search and returns a *SearchResults* object.
Returns *SearchResults* instance
For example:

```
>>> s = S().query(name__prefix='Jimmy')
>>> results = s.execute()
```

facet_counts()
Executes search and returns facet counts.

Example:

```
>>> s = S().query(name__prefix='Jimmy')
>>> facet_counts = s.facet_counts()
```

2.7.3 The F class

class elasticutils.**F**(**filters*)
Filter objects.

Makes it easier to create filters cumulatively using & (and), | (or) and ~ (not) operations.

For example:

```
f = F()
f &= F(price='Free')
f |= F(style='Mexican')
```

creates a filter "price = 'Free' or style = 'Mexican"'.

2.7.4 The Q class

class elasticutils.**Q**(**queries*)
Query objects.

Makes it easier to create queries cumulatively.

If there's more than one query part, they're combined under a BooleanQuery. By default, they're combined in the *must* clause.

You can combine two Q classes using the + operator. For example:

```
q = Q()
q += Q(title__text='shoes')
q += Q(summary__text='shoes')
```

creates a BooleanQuery with two *must* clauses.

Example 2:

```
q = Q()
q += Q(title__text='shoes', should=True)
q += Q(summary__text='shoes')
q += Q(description__text='shoes', must=True)
```

creates a BooleanQuery with one *should* clause (title) and two *must* clauses (summary and description).

2.7.5 The SearchResults class

class elasticutils.**SearchResults**(*type*, *response*, *results*, *fields*)

After executing a search, this is the class that manages the results.

Property type the mapping type of the S that created this SearchResults instance

Property took the amount of time the search took

Property count the total results

Property response the raw Elasticsearch search response

Property results the search results from the response if any

Property fields the list of fields specified by values_list or values_dict

When you iterate over this object, it returns the individual search results in the shape you asked for (object, tuple, dict, etc) in the order returned by Elasticsearch.

Example:

```
s = S().query(bio__text='archaeologist')
results = s.execute()

# Shows how long the search took
print results.took

# Shows the raw Elasticsearch response
print results.results
```

2.7.6 The MappingType class

class elasticutils.**MappingType**

Base class for mapping types.

To extend this class:

1. implement `get_index`.
2. implement `get_mapping_type_name`.
3. if this ties back to a model, implement `get_model` and possibly also `get_object`.

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)
```

classmethod **from_results**(*results_dict*)

get_object()
Returns the model instance

This gets called when someone uses the `.object` attribute which triggers lazy-loading of the object this document is based on.

By default, this calls:

```
self.get_model().get(id=self._id)
```

where `self._id` is the Elasticsearch document id.

Override it to do something different.

classmethod **get_index**()
Returns the index to use for this mapping type.

You can specify the index to use for this mapping type. This affects `S` built with this type.

By default, raises NotImplementedError.

Override this to return the index this mapping type should be indexed and searched in.

classmethod **get_mapping_type_name**()
Returns the mapping type name.

You can specify the mapping type name (also sometimes called the document type) with this method.

By default, raises NotImplementedError.

Override this to return the mapping type name.

classmethod **get_model**()
Return the model class related to this MappingType.

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

By default, raises NoModelError.

Override this to return a class that works with `.get_object()` to return the instance of the model that is related to this document.

2.7.7 The Indexable class

class elasticutils.**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 **bulk_index**(*documents*, *id_field='id'*, *es=None*, *index=None*)
Adds or updates a batch of documents.

Parameters

- **documents** – List of Python dicts representing individual documents to be added to the index

  Note: This must be serializable into JSON.

- **id_field** – The name of the field to use as the document id. This defaults to 'id'.
- **es** – The *ElasticSearch* to use. If you don't specify an *ElasticSearch*, it'll use *cls.get_es()*.
- **index** – The name of the index to use. If you don't specify one it'll use *cls.get_index()*.

Note: If you need the documents available for searches immediately, make sure to refresh the index by calling `refresh_index()`.

classmethod **`extract_document`**(*obj_id*, *obj=None*)
Extracts the Elasticsearch 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 object 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_es`**()
Returns an ElasticSearch object

Override this if you need special functionality.

Returns a pyelasticsearch *ElasticSearch* instance

classmethod **`get_indexable`**()
Returns an iterable of things to index.

Returns iterable of things to index

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

Example:

```
@classmethod
def get_mapping(cls):
    return {
        'properties': {
            'id': {'type': 'integer'},
            'name': {'type': 'string'}
        }
    }
```

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

Override this to return a mapping for this doctype.

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

classmethod **index**(*document*, *id_=None*, *force_insert=False*, *es=None*, *index=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 id of the document

  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.

- **force_insert** – TODO
- **es** – The *ElasticSearch* to use. If you don't specify an *ElasticSearch*, it'll use *cls.get_es()*.
- **index** – The name of the index to use. If you don't specify one it'll use *cls.get_index()*.

Note: If you need the documents available for searches immediately, make sure to refresh the index by calling `refresh_index()`.

---

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

Elasticsearch will update the index periodically automatically. If you need to see the documents you just indexed in your search results right now, you should call *refresh_index* as soon as you're done indexing. This is particularly helpful for unit tests.

Parameters

- **es** – The *ElasticSearch* to use. If you don't specify an *ElasticSearch*, it'll use *cls.get_es()*.
- **index** – The name of the index to use. If you don't specify one it'll use *cls.get_index()*.

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

Parameters

- **id** – The Elasticsearch id for the document to remove from the index.
- **es** – The *ElasticSearch* to use. If you don't specify an *ElasticSearch*, it'll use *cls.get_es()*.
- **index** – The name of the index to use. If you don't specify one it'll use *cls.get_index()*.

2.7.8 The DefaultMappingType class

class `elasticutils.`**DefaultMappingType**
This is the default mapping type for S.

2.7.9 The MLT class

class elasticutils.**MLT**(*id_*, *s=None*, *mlt_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*, *mlt_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. Allows you to pass in a query which will be used as the body of the more-like-this request.
- **mlt_fields** – A list of fields to look at for more like this.
- **index** – The index to use. Falls back to the first index listed in s.get_indexes().
- **doctype** – The doctype to use. Falls back to the first doctype listed in s.get_doctypes().
- **es** – The *ElasticSearch* 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*.

---

to_python(*obj*)

Converts strings in a data structure to Python types

It converts datetime-ish things to Python datetimes.

Override if you want something different.

Parameters **obj** – Python datastructure

Returns Python datastructure with strings converted to Python types

---

Note: This does the conversion in-place!

---

get_es()

Returns an *ElasticSearch*.

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

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

•Otherwise, it creates a new *ElasticSearch* 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.

CHAPTER

THREE

USING ELASTICUTILS WITH DJANGO

3.1 Using ElasticUtils with Django

- Summary
- How to integrate ElasticUtils with Django
- Configuration
- ElasticSearch
- Using with Django ORM models
- Celery tasks
- Middleware
- Writing tests
- Helpful things to know
  - Indexing and reset_queries

3.1.1 Summary

Django-specific code is all located in *elasticutils.contrib.django*.

This chapter covers using ElasticUtils Django bits. For API documentation, see *Django API docs*.

3.1.2 How to integrate ElasticUtils with Django

1. add ElasticUtils configuration settings to your project's setting file
2. write one or more *MappingType* classes
3. write code to create the Elasticsearch index and populate it with documents based on your *MappingType* subclasses
3. use `elasticutils.contrib.django.S` to search and return results
4. use `elasticutils.contrib.django.estestcase.ESTestCase` to write tests

That's the gist of it. You can deviate on any of these depending on your needs, of course.

3.1.3 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_URLS**
This is a list of Elasticsearch urls. In development this will look like:

```
ES_URLS = ['http://localhost: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**
Default: 5

The timeout in seconds for creating the Elasticsearch connection.

3.1.4 ElasticSearch

The *get_es()* in the Django contrib will use Django settings listed above to build the pyelasticsearch ElasticSearch object.

3.1.5 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` is *MappingType* which has some additional Django ORM-specific code in it to make it easier.

Define a *MappingType* 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 *MappingType* subclass:

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


class MyModel(Model):
    # Django model ...


class MyMappingType(MappingType):
    @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 import Indexable, MappingType


class MyModel(Model):
    # Django model ...


class MyMappingType(MappingType, 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 import Indexable, MappingType
```

```
class MyModel(Model):
    # Django model ...


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

    @classmethod
    def get_mapping(cls):
        """Returns an Elasticsearch mapping."""
        return {
            'properties': {
                # The id is an integer, so store it as such. Elasticsearch
                # 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()
```

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.

3.1.6 Celery tasks

Requirements Django, Celery

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

3.1.7 Middleware

Requirements Django

There's a middleware that catches all Elasticsearch-related exceptions and shows a 501/503 template accordingly. See `elasticutils.contrib.django.ESExceptionMiddleware` for details.

3.1.8 Writing tests

Requirements Django

When writing test cases for your ElasticUtils-using code, you'll want to do a few things:

1. Default `ES_DISABLED` to *True*. This way, the tests that kick off creating data but aren't testing search-specific things don't additionally index stuff. That'll save you a bunch of test time.
2. When testing ElasticUtils things, override the settings and set `ES_DISABLED` to *False*.
3. Use an `ESTestCase` that sets up the indexes before tests run and tears them down after they run.
4. When testing, make sure you use an index name that's unique. You don't want to run your tests and have them affect your production index.

You can use `elasticutils.contrib.django.estestcase.ESTestCase` for your app's tests. It's pretty basic but does all of the above except item 1 which you'll need to do in your test settings.

Example usage:

```
from elasticutils.contrib.django.estestcase import ESTestCase


class TestQueries(ESTestCase):
    # This class holds tests that do elasticsearch things

    def test_query(self):
        # Test code ...

    def test_locked_filters(self):
        # Test code ...
```

ElasticUtils uses this for it's Django tests. Look at the test code for more examples of usage:

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

If it's not what you want, you could subclass it and override behavior or just write your own.

3.1.9 Helpful things to know

Indexing and reset_queries

If you are:

1. indexing a lot of data pulled out with the Django ORM, and
2. have `DEBUG = True` (i.e. development environments)

then you'll probably want to call `django.db.reset_queries()` periodically.

What's going on is that when `DEBUG = True` (i.e. a devleopment environment), Django helpfully stores all the queries that are made which when you're indexing a lot of data is a lot of data. Calling `django.db.reset_queries()` periodically flushes the queries so it doesn't monotonically eat all your memory before the indexing is done.

3.2 Django API docs

- The S class
- The MappingType class
- The Indexable class
- View decorators
- The ESExceptionMiddleware class
- Tasks
- The ESTestCase class

3.2.1 The S class

class `elasticutils.contrib.django.`**`S`**(*mapping_type*)
S that's based on Django settings

This shows the Django-specific documentation. See `elasticutils.S` for more the rest.

`__init__`(*mapping_type*)
Create and return an S.

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

---

Note: The `elasticutils.S` doesn't require the *mapping_type* argument, but the `elasticutils.contrib.django.S` does.

`get_doctypes`(*default_doctypes=None*)
Returns the doctypes (or mapping type names) to use.

`get_es`(*default_builder=<function get_es at 0x3a6d398>*)
Returns the pyelasticsearch ElasticSearch object to use.

This uses the django get_es builder by default which takes into account settings in `settings.py`.

`get_indexes`(*default_indexes=None*)
Returns the list of indexes to act on based on ES_INDEXES setting

3.2.2 The MappingType class

class `elasticutils.contrib.django.`**`MappingType`**
MappingType that ties to Django ORM models

You probably want to subclass this and override at least *get_model()*.

This shows the Django-specific documentation. See `elasticutils.MappingType` for more the rest.

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

get_object()
Returns the database object for this result

By default, this is:

```
self.get_model().objects.get(pk=self._id)
```

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

Returns an *S* for this DjangoMappingType

3.2.3 The Indexable class

class `elasticutils.contrib.django.`**Indexable**
MappingType mixin that has indexing bits

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

This shows the Django-specific documentation. See `elasticutils.Indexable` for more the rest.

classmethod get_es(***overrides*)
Returns an ElasticSearch object using Django settings

Override this if you need special functionality.

Parameters **overrides** – Allows you to override defaults to create the ElasticSearch object. You can override any of the arguments listed in `elasticutils.get_es()`.

Returns a pyelasticsearch *ElasticSearch* instance

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

3.2.4 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 ElasticSearch instance to use.

elasticutils.contrib.django.**es_required_or_50x**(**m_args*, ***m_kwargs*)

3.2.5 The ESExceptionMiddleware class

class elasticutils.contrib.django.**ESExceptionMiddleware**(*disabled_template=None*, *error_template=None*)
Middleware to handle Elasticsearch errors.

HTTP 501 Returned when ES_DISABLED is True.

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

- pyelasticsearch.exceptions.ConnectionError
- pyelasticsearch.exceptions.ElasticHttpError
- pyelasticsearch.exceptions.ElasticHttpNotFoundError
- pyelasticsearch.exceptions.InvalidJsonResponseError
- pyelasticsearch.exceptions.Timeout

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.

Note: In order to use the included templates, you must add elasticutils.contrib.django to INSTALLED_APPS.

3.2.6 Tasks

elasticutils.contrib.django.tasks.**index_objects**(*model, ids=[...]*)
Index documents of a specified mapping type.

This allows for asynchronous indexing.

If a mapping_type extends Indexable, you can add a `post_save` hook for the model that it's based on like this:

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

Parameters

- **mapping_type** – the mapping type for these ids
- **ids** – the list of ids of things to index
- **chunk_size** – the size of the chunk for bulk indexing

Note: The default chunk_size is 100. The number of documents you can bulk index at once depends on the size of the documents.

3.2.7 The ESTestCase class

Subclass this and make it do what you need it to do. It's definitely worth reading the code.

class elasticutils.contrib.django.estestcase.**ESTestCase**(*methodName='runTest'*)
Test case scaffolding for ElasticUtils-using tests.

If `ES_URLS` is empty or missing or you can't connect to Elasticsearch specified in `ES_URLS`, then this will skip each individual test. This works with py.test, nose, and unittest in Python 2.7. If you don't have one of those, then this will print to stdout and just skip the test silently.

classmethod create_index(*index*, *settings=None*)
Creates index with given settings

Parameters

- **index** – the name of the index to create
- **settings** – dict of settings to set in *create_index* call

classmethod get_es()
Returns an ES

Override this if you need different settings for your ES.

classmethod index_data(*documents*, *index*, *doctype*, *id_field='id'*)
Bulk indexes given data.

This does a refresh after the data is indexed.

Parameters

- **documents** – list of python dicts each a document to index
- **index** – name of the index

- **doctype** – mapping type name
- **id_field** – the field the document id is stored in in the document

classmethod **refresh**(*index*)
Refresh index after indexing.

Parameters **index** – the name of the index to refresh. use `_all` to refresh all of them

setUp()
Skips the test if this class is skipping tests.

classmethod **setUpClass**()
Sets up the environment for ES tests

•pings the ES server—if this fails, it marks all the tests for skipping

•fixes settings

•deletes the test index if there is one

classmethod **tearDownClass**()
Tears down environment

•unfixes settings

•deletes the test index

CHAPTER

FOUR

CONTRIBUTOR'S GUIDE

4.1 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!

4.1.1 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*.

4.2 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*.

4.2.1 External requirements

You should have Elasticsearch installed and running.

4.2.2 Install dependencies

Run:

```
$ virtualenv ./venv
$ . ./venv/bin/activate
$ pip install -r requirements-dev.txt
$ python setup.py develop
```

This sets up 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.

4.3 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.

4.4 Documentation

4.4.1 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.

Also, of extreme high-priority hyperbole-ignoring importance:

ElasticSearch (camel-case)

Refers to the pyelasticsearch ElasticSearch class and instances.

Elasticsearch (no camel-case)

The Elasticsearch software.

4.4.2 Building the docs

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

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

4.5 Running and writing tests

4.5.1 Running the tests

You can run the tests with:

```
./run_tests.py
```

This will run all the 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.

4.5.2 Writing tests

Tests are located in *elasticutils/tests/*.

We use nose for test utilities and running tests.

4.5.3 ElasticTestCase

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

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

Superclass for Elasticsearch-using test cases.

Property index_name name of the index to use

Property mapping_type_name the mapping type name

Property es_settings settings to use to build a pyelasticsearch ElasticSearch object.

Property mapping the mapping to use when creating an index

Property data any data to add to the index in setup_class

Property skip_tests 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 Elasticsearch 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.

4.6 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) Run sample programs in docs.
   (d) 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

FIVE

SAMPLE PROGRAMS

5.1 Basic sample program

Here's a short script that gives you the gist of how to use ElasticUtils:

```
"""
This is a sample program that uses pyelasticsearch ElasticSearch
object 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

from pyelasticsearch.exceptions import ElasticHttpNotFoundError


URL = 'http://localhost:9200'
INDEX = 'fooindex'
DOCTYPE = 'testdoc'


# This creates a pyelasticsearch ElasticSearch object which we can use
# to do all our indexing.
es = get_es(urls=[URL])

# First, delete the index if it exists.
try:
    es.delete_index(INDEX)
except ElasticHttpNotFoundError:
    pass

# 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.
mapping = {
    DOCTYPE: {
        'properties': {
            'id': {'type': 'integer'},
            'title': {'type': 'string', 'analyzer': 'snowball'},
            'topics': {'type': 'string'},
            'product': {'type': 'string', 'index': 'not_analyzed'},
            }
        }
    }
```

```
40
41  # Create the index 'testdoc' mapping.
42  es.create_index(INDEX, settings={'mappings': mapping})
43
44
45  # Let's index some documents and make them available for searching.
46  documents = [
47      {'id': 1,
48       'title': 'Deleting cookies',
49       'topics': ['cookies', 'privacy'],
50       'product': ['Firefox', 'Firefox for mobile']},
51      {'id': 2,
52       'title': 'What is a cookie?',
53       'topics': ['cookies', 'privacy'],
54       'product': ['Firefox', 'Firefox for mobile']},
55      {'id': 3,
56       'title': 'Websites say cookies are blocked - Unblock them',
57       'topics': ['cookies', 'privacy', 'websites'],
58       'product': ['Firefox', 'Firefox for mobile', 'Boot2Gecko']},
59      {'id': 4,
60       'title': 'Awesome Bar',
61       'topics': ['tips', 'search', 'user interface'],
62       'product': ['Firefox']},
63      {'id': 5,
64       'title': 'Flash',
65       'topics': ['flash'],
66       'product': ['Firefox']}
67      ]
68
69  es.bulk_index(INDEX, DOCTYPE, documents, id_field='id')
70  es.refresh(INDEX)
71
72
73  # Now let's do some basic queries.
74
75  # Let's build a basic S that looks at our Elasticsearch cluster and
76  # the index and doctype we just indexed our documents in.
77  basic_s = S().es(urls=[URL]).indexes(INDEX).doctypes(DOCTYPE)
78
79  # How many documents are in our index?
80  print basic_s.count()
81  # Prints:
82  # 5
83
84  # Print articles with 'cookie' in the title.
85  print [item['title']
86         for item in basic_s.query(title__text='cookie')]
87  # Prints:
88  # [u'Deleting cookies', u'What is a cookie?',
89  #  u'Websites say cookies are blocked - Unblock them']
90
91  # Print articles with 'cookie' in the title that are related to
92  # websites.
93  print [item['title']
94         for item in basic_s.query(title__text='cookie')
95                            .filter(topics='websites')]
96  # Prints:
97  # [u'Websites say cookies are blocked - Unblock them']
```

```
98
99  # Print articles in the 'search' topic.
100 print [item['title']
101        for item in basic_s.filter(topics='search')]
102 # Prints:
103 # [u'Awesome Bar']
104
105 # Do a query and use the highlighter to denote the matching text.
106 print [(item['title'], item._highlight['title'])
107        for item in basic_s.query(title__text='cookie').highlight('title')]
108 # Prints:
109 # [
110 #     (u'Deleting cookies', [u'Deleting <em>cookies</em>']),
111 #     (u'What is a cookie?', [u'What is a <em>cookie</em>?']),
112 #     (u'Websites say cookies are blocked - Unblock them',
113 #        [u'Websites say <em>cookies</em> are blocked - Unblock them']
114 #     )
115 # ]
116
117
118 # That's the gist of it!
```

5.2 Sample program using facets

```
1  """
2  This is a sample program that uses pyelasticsearch ElasticSearch
3  object to create an index, create a mapping, and index some data. Then
4  it uses ElasticUtils S to show some behavior with facets.
5  """
6
7  from elasticutils import get_es, S
8
9  from pyelasticsearch.exceptions import ElasticHttpNotFoundError
10
11
12 URL = 'http://localhost:9200'
13 INDEX = 'fooindex'
14 DOCTYPE = 'testdoc'
15
16
17 # This creates a pyelasticsearch ElasticSearch object which we can use
18 # to do all our indexing.
19 es = get_es(urls=[URL])
20
21 # First, delete the index.
22 try:
23     es.delete_index(INDEX)
24 except ElasticHttpNotFoundError:
25     # Getting this here means the index doesn't exist, so there's
26     # nothing to delete.
27     pass
28
29 # Define the mapping for the doctype 'testdoc'. It's got an id field,
30 # a title which is analyzed, and two fields that are lists of tags, so
31 # we don't want to analyze them.
32 #
```

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


# This indexes a series of documents each is a Python dict.
documents = [
    {'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.bulk_index(INDEX, DOCTYPE, documents, id_field='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 Elasticsearch cluster,
# index and doctype.
basic_s = S().es(urls=[URL]).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'}
#     ]}

# Note that the facet_counts are affected by the query.

# Let's do a filter for 'flash' in the topic.
print s.filter(topics='flash').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'}
#     ]}

# Note that the facet_counts are NOT affected by filters.

# Let's do a filter for 'flash' in the topic, and specify
# filtered=True.
print s.facet('product', filtered=True).filter(topics='flash').facet_counts()
# Pretty-printed output:
# {u'product': [
#     {u'count': 1, u'term': u'Firefox'}
#     ]}

# Using filtered=True causes the facet_counts to be affected by the
# filters.

# We've done a bunch of faceting on a field that is not
# analyzed. Let's look at what happens when we try to use facets on a
# field that is analyzed.
print basic_s.facet('topics').facet_counts()
# Pretty-printed output:
# {u'topics': [
#     {u'count': 3, u'term': u'privacy'},
#     {u'count': 3, u'term': u'cookies'},
#     {u'count': 2, u'term': u'basic'},
#     {u'count': 1, u'term': u'websites'},
#     {u'count': 1, u'term': u'user'},
#     {u'count': 1, u'term': u'tips'},
#     {u'count': 1, u'term': u'search'},
#     {u'count': 1, u'term': u'interface'},
#     {u'count': 1, u'term': u'flash'}
#     ]}

# Note how the facet counts shows 'user' and 'interface' as two
```

```
149 # separate terms even though they're a single topic for document with
150 # id=4. When that document is indexed, the topic field is analyzed and
151 # the default analyzer tokenizes it splitting it into two terms.
152 #
153 # Moral of the story is that you want fields you facet on to be
154 # analyzed as keyword fields or not analyzed at all.
```

CHAPTER

SIX

INDICES AND TABLES

- *genindex*

PYTHON MODULE INDEX

d

django.conf.settings, ??

e

elasticutils.contrib.django.tasks, ??