Elasticsearch DSL Documentation

Release 6.1.0

Honza Král

Jan 09, 2018

Contents

1 Compatibility **3**

2 Search Example **5**

3 Persistence Example **7**

4 Pre-built Faceted Search **9**

5 Migration from `elasticsearch-py` **11**

6 License **13**

7 Contents **15**

7.1 Configuration 15
7.2 Search DSL 16
7.3 Persistence 24
7.4 Faceted Search 30
7.5 API Documentation 32
7.6 Contribution Guide 42
7.7 Changelog 42

Python Module Index **47**

Elasticsearch DSL is a high-level library whose aim is to help with writing and running queries against Elasticsearch. It is built on top of the official low-level client (`elasticsearch-py`).

It provides a more convenient and idiomatic way to write and manipulate queries. It stays close to the Elasticsearch JSON DSL, mirroring its terminology and structure. It exposes the whole range of the DSL from Python either directly using defined classes or a queryset-like expressions.

It also provides an optional wrapper for working with documents as Python objects: defining mappings, retrieving and saving documents, wrapping the document data in user-defined classes.

To use the other Elasticsearch APIs (eg. cluster health) just use the underlying client.

CHAPTER 1

Compatibility

The library is compatible with all Elasticsearch versions since `1.x` but you **have to use a matching major version**:

For **Elasticsearch 5.0** and later, use the major version 5 (`5.x.y`) of the library.

For **Elasticsearch 2.0** and later, use the major version 2 (`2.x.y`) of the library.

For **Elasticsearch 1.0** and later, use the major version 0 (`0.x.y`) of the library.

The recommended way to set your requirements in your *setup.py* or *requirements.txt* is:

```
# Elasticsearch 5.x
elasticsearch-dsl>=5.0.0,<6.0.0

# Elasticsearch 2.x
elasticsearch-dsl>=2.0.0,<3.0.0

# Elasticsearch 1.x
elasticsearch-dsl<2.0.0
```

The development is happening on `master` and `1.x` branches, respectively.

CHAPTER 2

Search Example

Let's have a typical search request written directly as a `dict`:

```
from elasticsearch import Elasticsearch
client = Elasticsearch()

response = client.search(
    index="my-index",
    body={
      "query": {
        "filtered": {
          "query": {
            "bool": {
              "must": [{"match": {"title": "python"}}],
              "must_not": [{"match": {"description": "beta"}}]
            }
          },
          "filter": {"term": {"category": "search"}}
        }
      },
      "aggs" : {
        "per_tag": {
          "terms": {"field": "tags"},
          "aggs": {
            "max_lines": {"max": {"field": "lines"}}
          }
        }
      }
    }
)

for hit in response['hits']['hits']:
    print(hit['_score'], hit['_source']['title'])

for tag in response['aggregations']['per_tag']['buckets']:
    print(tag['key'], tag['max_lines']['value'])
```

The problem with this approach is that it is very verbose, prone to syntax mistakes like incorrect nesting, hard to modify (eg. adding another filter) and definitely not fun to write.

Let's rewrite the example using the Python DSL:

```
from elasticsearch import Elasticsearch
from elasticsearch_dsl import Search

client = Elasticsearch()

s = Search(using=client, index="my-index") \
    .filter("term", category="search") \
    .query("match", title="python")   \
    .exclude("match", description="beta")

s.aggs.bucket('per_tag', 'terms', field='tags') \
    .metric('max_lines', 'max', field='lines')

response = s.execute()

for hit in response:
    print(hit.meta.score, hit.title)

for tag in response.aggregations.per_tag.buckets:
    print(tag.key, tag.max_lines.value)
```

As you see, the library took care of:

- creating appropriate `Query` objects by name (eq. "match")
- composing queries into a compound `bool` query
- creating a `filtered` query since `.filter()` was used
- providing a convenient access to response data
- no curly or square brackets everywhere

CHAPTER 3

Persistence Example

Let's have a simple Python class representing an article in a blogging system:

```
from datetime import datetime
from elasticsearch_dsl import DocType, Date, Integer, Keyword, Text
from elasticsearch_dsl.connections import connections

# Define a default Elasticsearch client
connections.create_connection(hosts=['localhost'])

class Article(DocType):
    title = Text(analyzer='snowball', fields={'raw': Keyword()})
    body = Text(analyzer='snowball')
    tags = Keyword()
    published_from = Date()
    lines = Integer()

    class Meta:
        index = 'blog'

    def save(self, ** kwargs):
        self.lines = len(self.body.split())
        return super(Article, self).save(** kwargs)

    def is_published(self):
        return datetime.now() >= self.published_from

# create the mappings in elasticsearch
Article.init()

# create and save and article
article = Article(meta={'id': 42}, title='Hello world!', tags=['test'])
article.body = ''' looong text '''
article.published_from = datetime.now()
article.save()
```

```
article = Article.get(id=42)
print(article.is_published())

# Display cluster health
print(connections.get_connection().cluster.health())
```

In this example you can see:

- providing a *Default connection*
- defining fields with mapping configuration
- setting index name
- defining custom methods
- overriding the built-in `.save()` method to hook into the persistence life cycle
- retrieving and saving the object into Elasticsearch
- accessing the underlying client for other APIs

You can see more in the *Persistence* chapter.

CHAPTER 4

Pre-built Faceted Search

If you have your `DocTypes` defined you can very easily create a faceted search class to simplify searching and filtering.

Note: This feature is experimental and may be subject to change.

```
from elasticsearch_dsl import FacetedSearch
from elasticsearch_dsl.aggs import Terms, DateHistogram

class BlogSearch(FacetedSearch):
    doc_types = [Article, ]
    # fields that should be searched
    fields = ['tags', 'title', 'body']

    facets = {
        # use bucket aggregations to define facets
        'tags': Terms(field='tags'),
        'publishing_frequency': DateHistogram(field='published_from', interval='month
↪')
    }

# empty search
bs = BlogSearch()
response = bs.execute()

for hit in response:
    print(hit.meta.score, hit.title)

for (tag, count, selected) in response.facets.tags:
    print(tag, ' (SELECTED):' if selected else ':', count)

for (month, count, selected) in response.facets.publishing_frequency:
    print(month.strftime('%B %Y'), ' (SELECTED):' if selected else ':', count)
```

You can find more details in the *Faceted Search* chapter.

CHAPTER 5

Migration from `elasticsearch-py`

You don't have to port your entire application to get the benefits of the Python DSL, you can start gradually by creating a `Search` object from your existing `dict`, modifying it using the API and serializing it back to a `dict`:

```
body = {...} # insert complicated query here

# Convert to Search object
s = Search.from_dict(body)

# Add some filters, aggregations, queries, ...
s.filter("term", tags="python")

# Convert back to dict to plug back into existing code
body = s.to_dict()
```

CHAPTER 6

License

Copyright 2013 Elasticsearch

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

CHAPTER 7

Contents

7.1 Configuration

There are several ways how to configure connections for the library. Easiest option, and most useful, is to just define one default connection that will be used every time an API call is made without explicitly passing in other connection.

When using `elasticsearch_dsl` it is highly recommended to use the attached serializer (`elasticsearch_dsl.serializer.serializer`) that will make sure your objects are correctly serialized into json every time. The `create_connection` method that is described here (and that `configure` method uses under the hood) will do that automatically for you, unless you explicitly specify your own serializer. The serializer we use will also allow you to serialize your own objects - just define a `to_dict()` method on your objects and it will automatically be called when serializing to json.

Note: Unless you want to access multiple clusters from your application it is highly recommended that you use the `create_connection` method and all operations will use that connection automatically.

7.1.1 Manual

If you don't wish to supply global configuration you can always pass in your own connection (instance of `elasticsearch.Elasticsearch`) as parameter `using` wherever it is accepted:

```
s = Search(using=Elasticsearch('localhost'))
```

You can even use this approach to override any connection the object might be already associated with:

```
s = s.using(Elasticsearch('otherhost:9200'))
```

7.1.2 Default connection

To define a default connection that will be used globally, use the `connections` module and the `create_connection` method:

```
from elasticsearch_dsl import connections

connections.create_connection(hosts=['localhost'], timeout=20)
```

Any keyword arguments (`hosts` and `timeout` in our example) will be passed to the `Elasticsearch` class from `elasticsearch-py`. To see all the possible configuration options see the documentation.

7.1.3 Multiple clusters

You can define multiple connections to multiple clusters, either at the same time using the `configure` method:

```
from elasticsearch_dsl import connections

connections.configure(
    default={'hosts': 'localhost'},
    dev={
        'hosts': ['esdev1.example.com:9200'],
        'sniff_on_start': True
    }
)
```

Such connections will be constructed lazily when requested for the first time.

Or just add them one by one:

```
# if you have configuration to be passed to Elasticsearch.__init__
connections.create_connection('qa', hosts=['esqa1.example.com'], sniff_on_start=True)

# if you already have an Elasticsearch instance ready
connections.add_connection('qa', my_client)
```

Using aliases

When using multiple connections you can just refer to them using the string alias you registered them under:

```
s = Search(using='qa')
```

`KeyError` will be raised if there is no connection registered under that alias.

7.2 Search DSL

7.2.1 The `Search` object

The `Search` object represents the entire search request:

- queries
- filters

- aggregations
- sort
- pagination
- additional parameters
- associated client

The API is designed to be chainable. With the exception of the aggregations functionality this means that the `Search` object is immutable - all changes to the object will result in a copy being created which contains the changes. This means you can safely pass the `Search` object to foreign code without fear of it modifying your objects.

You can pass an instance of the low-level elasticsearch client when instantiating the `Search` object:

```
from elasticsearch import Elasticsearch
from elasticsearch_dsl import Search

client = Elasticsearch()

s = Search(using=client)
```

You can also define the client at a later time (for more options see the ~:ref:*connections* chapter):

```
s = s.using(client)
```

Note: All methods return a *copy* of the object, making it safe to pass to outside code.

The API is chainable, allowing you to combine multiple method calls in one statement:

```
s = Search().using(client).query("match", title="python")
```

To send the request to Elasticsearch:

```
response = s.execute()
```

If you just want to iterate over the hits returned by your search you can iterate over the `Search` object:

```
for hit in s:
    print(hit.title)
```

Search results will be cached. Subsequent calls to `execute` or trying to iterate over an already executed `Search` object will not trigger additional requests being sent to Elasticsearch. To force a request specify `ignore_cache=True` when calling `execute`.

For debugging purposes you can serialize the `Search` object to a `dict` explicitly:

```
print(s.to_dict())
```

Delete By Query

You can delete the documents matching a search by calling `delete` on the `Search` object instead of `execute` like this:

```
s = Search().query("match", title="python")
response = s.delete()
```

Queries

The library provides classes for all Elasticsearch query types. Pass all the parameters as keyword arguments. The classes accept any keyword arguments, the dsl then takes all arguments passed to the constructor and serializes them as top-level keys in the resulting dictionary (and thus the resulting json being sent to elasticsearch). This means that there is a clear one-to-one mapping between the raw query and its equivalent in the DSL:

```
from elasticsearch_dsl.query import MultiMatch, Match

# {"multi_match": {"query": "python django", "fields": ["title", "body"]}}
MultiMatch(query='python django', fields=['title', 'body'])

# {"match": {"title": {"query": "web framework", "type": "phrase"}}}
Match(title={"query": "web framework", "type": "phrase"})
```

Note: In some cases this approach is not possible due to python's restriction on identifiers - for example if your field is called `@timestamp`. In that case you have to fall back to unpacking a dictionary: `Range(** {'@timestamp': {'lt': 'now'}})`

You can use the `Q` shortcut to construct the instance using a name with parameters or the raw `dict`:

```
Q("multi_match", query='python django', fields=['title', 'body'])
Q({"multi_match": {"query": "python django", "fields": ["title", "body"]}})
```

To add the query to the `Search` object, use the `.query()` method:

```
q = Q("multi_match", query='python django', fields=['title', 'body'])
s = s.query(q)
```

The method also accepts all the parameters as the `Q` shortcut:

```
s = s.query("multi_match", query='python django', fields=['title', 'body'])
```

If you already have a query object, or a `dict` representing one, you can just override the query used in the `Search` object:

```
s.query = Q('bool', must=[Q('match', title='python'), Q('match', body='best')])
```

Query combination

Query objects can be combined using logical operators:

```
Q("match", title='python') | Q("match", title='django')
# {"bool": {"should": [...]}}

Q("match", title='python') & Q("match", title='django')
# {"bool": {"must": [...]}}

~Q("match", title="python")
# {"bool": {"must_not": [...]}}
```

When you call the `.query()` method multiple times, the `&` operator will be used internally:

```
s = s.query().query()
print(s.to_dict())
# {"query": {"bool": {...}}}
```

If you want to have precise control over the query form, use the `Q` shortcut to directly construct the combined query:

```
q = Q('bool',
    must=[Q('match', title='python')],
    should=[Q(...), Q(...)],
    minimum_should_match=1
)
s = Search().query(q)
```

Filters

If you want to add a query in a filter context you can use the `filter()` method to make things easier:

```
s = Search()
s = s.filter('terms', tags=['search', 'python'])
```

Behind the scenes this will produce a `Bool` query and place the specified `terms` query into its `filter` branch, making it equivalent to:

```
s = Search()
s = s.query('bool', filter=[Q('terms', tags=['search', 'python'])])
```

If you want to use the post_filter element for faceted navigation, use the `.post_filter()` method.

You can also `exclude()` items from your query like this:

```
s = Search()
s = s.exclude('terms', tags=['search', 'python'])
```

which is shorthand for: `s = s.query('bool', filter=[~Q('terms', tags=['search', 'python'])])`

Aggregations

To define an aggregation, you can use the `A` shortcut:

```
A('terms', field='tags')
# {"terms": {"field": "tags"}}
```

To nest aggregations, you can use the `.bucket()`, `.metric()` and `.pipeline()` methods:

```
a = A('terms', field='category')
# {'terms': {'field': 'category'}}

a.metric('clicks_per_category', 'sum', field='clicks')\
    .bucket('tags_per_category', 'terms', field='tags')
# {
#   'terms': {'field': 'category'},
#   'aggs': {
#     'clicks_per_category': {'sum': {'field': 'clicks'}},
#     'tags_per_category': {'terms': {'field': 'tags'}}
```

```
#   }
# }
```

To add aggregations to the `Search` object, use the `.aggs` property, which acts as a top-level aggregation:

```
s = Search()
a = A('terms', field='category')
s.aggs.bucket('category_terms', a)
# {
#   'aggs': {
#     'category_terms': {
#       'terms': {
#         'field': 'category'
#       }
#     }
#   }
# }
```

or

```
s = Search()
s.aggs.bucket('articles_per_day', 'date_histogram', field='publish_date', interval=
↪'day')\
    .metric('clicks_per_day', 'sum', field='clicks')\
    .pipeline('moving_click_average', 'moving_avg', buckets_path='clicks_per_day')\
    .bucket('tags_per_day', 'terms', field='tags')

s.to_dict()
# {
#   "aggs": {
#     "articles_per_day": {
#       "date_histogram": { "interval": "day", "field": "publish_date" },
#       "aggs": {
#         "clicks_per_day": { "sum": { "field": "clicks" } },
#         "moving_click_average": { "moving_avg": { "buckets_path": "clicks_per_day" }
↪ },
#         "tags_per_day": { "terms": { "field": "tags" } }
#       }
#     }
#   }
# }
```

You can access an existing bucket by its name:

```
s = Search()

s.aggs.bucket('per_category', 'terms', field='category')
s.aggs['per_category'].metric('clicks_per_category', 'sum', field='clicks')
s.aggs['per_category'].bucket('tags_per_category', 'terms', field='tags')
```

Note: When chaining multiple aggregations, there is a difference between what `.bucket()` and `.metric()` methods return - `.bucket()` returns the newly defined bucket while `.metric()` returns its parent bucket to allow further chaining.

As opposed to other methods on the `Search` objects, defining aggregations is done in-place (does not return a copy).

Sorting

To specify sorting order, use the `.sort()` method:

```
s = Search().sort(
    'category',
    '-title',
    {"lines" : {"order" : "asc", "mode" : "avg"}}
)
```

It accepts positional arguments which can be either strings or dictionaries. String value is a field name, optionally prefixed by the − sign to specify a descending order.

To reset the sorting, just call the method with no arguments:

```
s = s.sort()
```

Pagination

To specify the from/size parameters, use the Python slicing API:

```
s = s[10:20]
# {"from": 10, "size": 10}
```

If you want to access all the documents matched by your query you can use the `scan` method which uses the scan/scroll elasticsearch API:

```
for hit in s.scan():
    print(hit.title)
```

Note that in this case the results won't be sorted.

Highlighting

To set common attributes for highlighting use the `highlight_options` method:

```
s = s.highlight_options(order='score')
```

Enabling highlighting for individual fields is done using the `highlight` method:

```
s = s.highlight('title')
# or, including parameters:
s = s.highlight('title', fragment_size=50)
```

The fragments in the response will then be available on reach `Result` object as `.meta.highlight.FIELD` which will contain the list of fragments:

```
response = s.execute()
for hit in response:
    for fragment in hit.meta.highlight.title:
        print(fragment)
```

Suggestions

To specify a suggest request on your `Search` object use the `suggest` method:

```
s = s.suggest('my_suggestion', 'pyhton', term={'field': 'title'})
```

The first argument is the name of the suggestions (name under which it will be returned), second is the actual text you wish the suggester to work on and the keyword arguments will be added to the suggest's json as-is which means that it should be one of `term`, `phrase` or `completion` to indicate which type of suggester should be used.

If you only wish to run the suggestion part of the search (via the `_suggest` endpoint) you can do so via `execute_suggest`:

```
s = s.suggest('my_suggestion', 'pyhton', term={'field': 'title'})
suggestions = s.execute_suggest()

print(suggestions.my_suggestion)
```

Extra properties and parameters

To set extra properties of the search request, use the `.extra()` method. This can be used to define keys in the body that cannot be defined via a specific API method like `explain` or `search_after`:

```
s = s.extra(explain=True)
```

To set query parameters, use the `.params()` method:

```
s = s.params(search_type="count")
```

If you need to limit the fields being returned by elasticsearch, use the `source()` method:

```
# only return the selected fields
s = s.source(['title', 'body'])
# don't return any fields, just the metadata
s = s.source(False)
# explicitly include/exclude fields
s = s.source(include=["title"], exclude=["user.*"])
# reset the field selection
s = s.source(None)
```

Serialization and Deserialization

The search object can be serialized into a dictionary by using the `.to_dict()` method.

You can also create a `Search` object from a `dict` using the `from_dict` class method. This will create a new `Search` object and populate it using the data from the dict:

```
s = Search.from_dict({"query": {"match": {"title": "python"}}})
```

If you wish to modify an existing `Search` object, overriding it's properties, instead use the `update_from_dict` method that alters an instance **in-place**:

```
s = Search(index='i')
s.update_from_dict({"query": {"match": {"title": "python"}}, "size": 42})
```

7.2.2 Response

You can execute your search by calling the `.execute()` method that will return a `Response` object. The `Response` object allows you access to any key from the response dictionary via attribute access. It also provides some convenient helpers:

```
response = s.execute()

print(response.success())
# True

print(response.took)
# 12

print(response.hits.total)

print(response.suggest.my_suggestions)
```

If you want to inspect the contents of the `response` objects, just use its `to_dict` method to get access to the raw data for pretty printing.

Hits

To access to the hits returned by the search, access the `hits` property or just iterate over the `Response` object:

```
response = s.execute()
print('Total %d hits found.' % response.hits.total)
for h in response:
    print(h.title, h.body)
```

Result

The individual hits is wrapped in a convenience class that allows attribute access to the keys in the returned dictionary. All the metadata for the results are accessible via `meta` (without the leading _):

```
response = s.execute()
h = response.hits[0]
print('/%s/%s/%s returned with score %f' % (
    h.meta.index, h.meta.doc_type, h.meta.id, h.meta.score))
```

Note: If your document has a field called `meta` you have to access it using the get item syntax: `hit['meta']`.

Aggregations

Aggregations are available through the `aggregations` property:

```
for tag in response.aggregations.per_tag.buckets:
    print(tag.key, tag.max_lines.value)
```

7.2.3 MultiSearch

If you need to execute multiple searches at the same time you can use the `MultiSearch` class which will use the `_msearch` API:

```
from elasticsearch_dsl import MultiSearch, Search

ms = MultiSearch(index='blogs')

ms = ms.add(Search().filter('term', tags='python'))
ms = ms.add(Search().filter('term', tags='elasticsearch'))

responses = ms.execute()

for response in responses:
    print("Results for query %r." % response.search.query)
    for hit in response:
        print(hit.title)
```

7.3 Persistence

You can use the dsl library to define your mappings and a basic persistent layer for your application.

7.3.1 Mappings

If you wish to create mappings manually you can use the `Mapping` class, for more advanced use cases, however, we recommend you use the *DocType* abstraction in combination with *Index* (or `IndexTemplate`) to define index-level settings and properties. The mapping definition follows a similar pattern to the query dsl:

```
from elasticsearch_dsl import Keyword, Mapping, Nested, Text

# name your type
m = Mapping('my-type')

# add fields
m.field('title', 'text')

# you can use multi-fields easily
m.field('category', 'text', fields={'raw': Keyword()})

# you can also create a field manually
comment = Nested(
                properties={
                    'author': Text(),
                    'created_at': Date()
                })

# and attach it to the mapping
m.field('comments', comment)

# you can also define mappings for the meta fields
m.meta('_all', enabled=False)
```

```
# save the mapping into index 'my-index'
m.save('my-index')
```

Note: By default all fields (with the exception of `Nested`) will expect single values. You can always override this expectation during the field creation/definition by passing in `multi=True` into the constructor (`m.field('tags', Keyword(multi=True))`). Then the value of the field, even if the field hasn't been set, will be an empty list enabling you to write `doc.tags.append('search')`.

Especially if you are using dynamic mappings it might be useful to update the mapping based on an existing type in Elasticsearch, or create the mapping directly from an existing type:

```
# get the mapping from our production cluster
m = Mapping.from_es('my-index', 'my-type', using='prod')

# update based on data in QA cluster
m.update_from_es('my-index', using='qa')

# update the mapping on production
m.save('my-index', using='prod')
```

Common field options:

`multi` If set to `True` the field's value will be set to `[]` at first access.

`required` Indicates if a field requires a value for the document to be valid.

7.3.2 Analysis

To specify `analyzer` values for `Text` fields you can just use the name of the analyzer (as a string) and either rely on the analyzer being defined (like built-in analyzers) or define the analyzer yourself manually.

Alternatively you can create your own analyzer and have the persistence layer handle its creation:

```
from elasticsearch_dsl import analyzer, tokenizer

my_analyzer = analyzer('my_analyzer',
    tokenizer=tokenizer('trigram', 'nGram', min_gram=3, max_gram=3),
    filter=['lowercase']
)
```

Each analysis object needs to have a name (`my_analyzer` and `trigram` in our example) and tokenizers, token filters and char filters also need to specify type (`nGram` in our example).

Note: When creating a mapping which relies on a custom analyzer the index must either not exist or be closed. To create multiple `DocType`-defined mappings you can use the *Index* object.

7.3.3 DocType

If you want to create a model-like wrapper around your documents, use the `DocType` class:

```
from datetime import datetime
from elasticsearch_dsl import DocType, Date, Nested, Boolean, \
    analyzer, InnerDoc, Completion, Keyword, Text

html_strip = analyzer('html_strip',
    tokenizer="standard",
    filter=["standard", "lowercase", "stop", "snowball"],
    char_filter=["html_strip"]
)

class Comment(InnerDoc):
    author = Text(fields={'raw': Keyword()})
    content = Text(analyzer='snowball')
    created_at = Date()

    def age(self):
        return datetime.now() - self.created_at

class Post(DocType):
    title = Text()
    title_suggest = Completion()
    created_at = Date()
    published = Boolean()
    category = Text(
        analyzer=html_strip,
        fields={'raw': Keyword()}
    )

    comments = Nested(Comment)

    class Meta:
        index = 'blog'

    def add_comment(self, author, content):
        self.comments.append(
          Comment(author=author, content=content, created_at=datetime.now()))

    def save(self, ** kwargs):
        self.created_at = datetime.now()
        return super().save(** kwargs)
```

Note on dates

`elasticsearch-dsl` will always respect the timezone information (or lack thereof) on the `datetime` objects passed in or stored in Elasticsearch. Elasticsearch itself interprets all datetimes with no timezone information as `UTC`. If you wish to reflect this in your python code, you can specify `default_timezone` when instantiating a `Date` field:

```
class Post(DocType):
    created_at = Date(default_timezone='UTC')
```

In that case any `datetime` object passed in (or parsed from elasticsearch) will be treated as if it were in `UTC` timezone.

Document life cycle

Before you first use the `Post` document type, you need to create the mappings in Elasticsearch. For that you can either use the *Index* object or create the mappings directly by calling the `init` class method:

```
# create the mappings in Elasticsearch
Post.init()
```

To create a new `Post` document just instantiate the class and pass in any fields you wish to set, you can then use standard attribute setting to change/add more fields. Note that you are not limited to the fields defined explicitly:

```
# instantiate the document
first = Post(title='My First Blog Post, yay!', published=True)
# assign some field values, can be values or lists of values
first.category = ['everything', 'nothing']
# every document has an id in meta
first.meta.id = 47


# save the document into the cluster
first.save()
```

All the metadata fields (`id`, `routing`, `index` etc) can be accessed (and set) via a `meta` attribute or directly using the underscored variant:

```
post = Post(meta={'id': 42})

# prints 42, same as post._id
print(post.meta.id)

# override default index, same as post._index
post.meta.index = 'my-blog'
```

Note: Having all metadata accessible through `meta` means that this name is reserved and you shouldn't have a field called `meta` on your document. If you, however, need it you can still access the data using the get item (as opposed to attribute) syntax: `post['meta']`.

To retrieve an existing document use the `get` class method:

```
# retrieve the document
first = Post.get(id=42)
# now we can call methods, change fields, ...
first.add_comment('me', 'This is nice!')
# and save the changes into the cluster again
first.save()

# you can also update just individual fields which will call the update API
# and also update the document in place
first.update(published=True, published_by='me')
```

If the document is not found in elasticsearch an exception (`elasticsearch.NotFoundError`) will be raised. If you wish to return `None` instead just pass in `ignore=404` to suppress the exception:

```
p = Post.get(id='not-in-es', ignore=404)
p is None
```

When you wish to retrive multiple documents at the same time by their `id` you can use the `mget` method:

```
posts = Post.mget([42, 47, 256])
```

`mget` will, by default, raise a `NotFoundError` if any of the documents wasn't found and `RequestError` if any of the document had resulted in error. You can control this behavior by setting parameters:

`raise_on_error` If `True` (default) then any error will cause an exception to be raised. Otherwise all documents containing errors will be treated as missing.

`missing` Can have three possible values: `'none'` (default), `'raise'` and `'skip'`. If a document is missing or errored it will either be replaced with `None`, an exception will be raised or the document will be skipped in the output list entirely.

All the information about the `DocType`, including its `Mapping` can be accessed through the `_doc_type` attribute of the class:

```
# name of the index in elasticsearch
Post._doc_type.index

# the raw Mapping object
Post._doc_type.mapping
```

The `_doc_type` attribute is also home to the `refresh` method which will update the mapping on the `DocType` from elasticsearch. This is very useful if you use dynamic mappings and want the class to be aware of those fields (for example if you wish the `Date` fields to be properly (de)serialized):

```
Post._doc_type.refresh()
```

To delete a document just call its `delete` method:

```
first = Post.get(id=42)
first.delete()
```

Search

To search for this document type, use the `search` class method:

```
# by calling .search we get back a standard Search object
s = Post.search()
# the search is already limited to the index and doc_type of our document
s = s.filter('term', published=True).query('match', title='first')


results = s.execute()

# when you execute the search the results are wrapped in your document class (Post)
for post in results:
    print(post.meta.score, post.title)
```

Alternatively you can just take a `Search` object and restrict it to return our document type, wrapped in correct class:

```
s = Search()
s = s.doc_type(Post)
```

You can also combine document classes with standard doc types (just strings), which will be treated as before. You can also pass in multiple `DocType` subclasses and each document in the response will be wrapped in it's class.

If you want to run suggestions, just use the `suggest` method on the `Search` object:

```
s = Post.search()
s = s.suggest('title_suggestions', 'pyth', completion={'field': 'title_suggest'})

# you can even execute just the suggestions via the _suggest API
suggestions = s.execute_suggest()

for result in suggestions.title_suggestions:
    print('Suggestions for %s:' % result.text)
    for option in result.options:
        print('  %s (%r)' % (option.text, option.payload))
```

`class Meta` options

In the `Meta` class inside your document definition you can define various metadata for your document:

`doc_type` name of the doc_type in elasticsearch. By default it will be set to `doc`, it is not recommended to change.

`index` default index for the document, by default it is empty and every operation such as `get` or `save` requires an explicit `index` parameter

`using` default connection alias to use, defaults to `'default'`

`mapping` optional instance of `Mapping` class to use as base for the mappings created from the fields on the document class itself.

`matches(self, hit)` method that returns `True` if a given raw hit (`dict` returned from elasticsearch) should be deserialized using this `DocType` subclass. Can be overriden, by default will just check that values for `_index` (including any wildcard expansions) and `_type` in the document matches those in `_doc_type`.

Any attributes on the `Meta` class that are instance of `MetaField` will be used to control the mapping of the meta fields (`_all`, `dynamic` etc). Just name the parameter (without the leading underscore) as the field you wish to map and pass any parameters to the `MetaField` class:

```
class Post(DocType):
    title = Text()

    class Meta:
        all = MetaField(enabled=False)
        dynamic = MetaField('strict')
```

7.3.4 Index

`Index` is a class responsible for holding all the metadata related to an index in elasticsearch - mappings and settings. It is most useful when defining your mappings since it allows for easy creation of multiple mappings at the same time. This is especially useful when setting up your elasticsearch objects in a migration:

```
from elasticsearch_dsl import Index, DocType, Text, analyzer

blogs = Index('blogs')

# define custom settings
blogs.settings(
    number_of_shards=1,
    number_of_replicas=0
)
```

```
# define aliases
blogs.aliases(
    old_blogs={}
)

# register a doc_type with the index
blogs.doc_type(Post)

# can also be used as class decorator when defining the DocType
@blogs.doc_type
class Post(DocType):
    title = Text()

# You can attach custom analyzers to the index

html_strip = analyzer('html_strip',
    tokenizer="standard",
    filter=["standard", "lowercase", "stop", "snowball"],
    char_filter=["html_strip"]
)

blogs.analyzer(html_strip)

# delete the index, ignore if it doesn't exist
blogs.delete(ignore=404)

# create the index in elasticsearch
blogs.create()
```

You can also set up a template for your indices and use the `clone` method to create specific copies:

```
blogs = Index('blogs', using='production')
blogs.settings(number_of_shards=2)
blogs.doc_type(Post)

# create a copy of the index with different name
company_blogs = blogs.clone('company-blogs')

# create a different copy on different cluster
dev_blogs = blogs.clone('blogs', using='dev')
# and change its settings
dev_blogs.setting(number_of_shards=1)
```

7.4 Faceted Search

The library comes with a simple abstraction aimed at helping you develop faceted navigation for your data.

Note: This API is experimental and will be subject to change. Any feedback is welcome.

7.4.1 Configuration

You can provide several configuration options (as class attributes) when declaring a `FacetedSearch` subclass:

`index` the name of the index (as string) to search through, defaults to `'_all'`.

`doc_types` list of `DocType` subclasses or strings to be used, defaults to `['_all']`.

`fields` list of fields on the document type to search through. The list will be passes to `MultiMatch` query so can contain boost values (`'title^5'`), defaults to `['*']`.

`facets` dictionary of facets to display/filter on. The key is the name displayed and values should be instances of any `Facet` subclass, for example: `{'tags': TermsFacet(field='tags')}`

`sort` tuple or list of fields on which the results should be sorted. The format of the individual fieldsa re to be the same as those passed to *`sort()`*.

Facets

There are several different facets available:

`TermsFacet` provides an option to split documents into groups based on a value of a field, for example `TermsFacet(field='category')`

`DateHistogramFacet` split documents into time intervals, example: `DateHistogramFacet(field="published_date", interval="day")`

`HistogramFacet` similar to `DateHistogramFacet` but for numerical values: `HistogramFacet(field="rating", interval=2)`

`RangeFacet` allows you to define your own ranges for a numerical fields: `RangeFacet(field="comment_count", ranges=[("few", (None, 2)), ("lots", (2, None))])`

Advanced

If you require any custom behavior or modifications simply override one or more of the methods responsible for the class' functions:

`search(self)` is responsible for constructing the `Search` object used. Override this if you want to customize the search object (for example by adding a global filter for published articles only).

`query(self, search)` adds the query postion of the search (if search input specified), by default using `MultiField` query. Override this if you want to modify the query type used.

`highlight(self, search)` defines the highlighting on the `Search` object and returns a new one. Default behavior is to highlight on all fields specified for search.

7.4.2 Usage

The custom subclass can be instantiated empty to provide an empty search (matching everything) or with `query` and `filters`.

`query` is used to pass in the text of the query to be performed. If `None` is passed in (default) a `MatchAll` query will be used. For example `'python web'`

`filters` is a dictionary containing all the facet filters that you wish to apply. Use the name of the facet (from `.facets` attribute) as the key and one of the possible values as value. For example `{'tags': 'python'}`.

Response

the response returned from the `FacetedSearch` object (by calling `.execute()`) is a subclass of the standard `Response` class that adds a property called `facets` which contains a dictionary with lists of buckets - each represented by a tuple of key, document count and a flag indicating whether this value has been filtered on.

7.4.3 Example

```
from datetime import date

from elasticsearch_dsl import FacetedSearch, TermsFacet, DateHistogramFacet

class BlogSearch(FacetedSearch):
    doc_types = [Article, ]
    # fields that should be searched
    fields = ['tags', 'title', 'body']

    facets = {
        # use bucket aggregations to define facets
        'tags': TermsFacet(field='tags'),
        'publishing_frequency': DateHistogramFacet(field='published_from', interval=
↪'month')
    }

    def search(self):
        # override methods to add custom pieces
        s = super().search()
        return s.filter('range', publish_from={'lte': 'now/h'})

bs = BlogSearch('python web', {'publishing_frequency': date(2015, 6)})
response = bs.execute()

# access hits and other attributes as usual
print(response.hits.total, 'hits total')
for hit in response:
    print(hit.meta.score, hit.title)

for (tag, count, selected) in response.facets.tags:
    print(tag, ' (SELECTED):' if selected else ':', count)

for (month, count, selected) in response.facets.publishing_frequency:
    print(month.strftime('%B %Y'), ' (SELECTED):' if selected else ':', count)
```

7.5 API Documentation

Below please find the documentation for the public classes and functions of `elasticsearch_dsl`.

7.5.1 Search

class elasticsearch_dsl.**Search**(***kwargs*)

Search request to elasticsearch.

Parameters

- **using** – *Elasticsearch* instance to use
- **index** – limit the search to index
- **doc_type** – only query this type.

All the parameters supplied (or omitted) at creation type can be later overriden by methods (*using*, *index* and *doc_type* respectively).

count()
Return the number of hits matching the query and filters. Note that only the actual number is returned.

delete() *executes the query by delegating to delete_by_query()*

execute(*ignore_cache=False*)
Execute the search and return an instance of `Response` wrapping all the data.

Parameters **response_class** – optional subclass of `Response` to use instead.

classmethod from_dict(*d*)
Construct a new *Search* instance from a raw dict containing the search body. Useful when migrating from raw dictionaries.

Example:

```
s = Search.from_dict({
    "query": {
        "bool": {
            "must": [...]
        }
    },
    "aggs": {...}
})
s = s.filter('term', published=True)
```

highlight(**fields*, ***kwargs*)
Request highlighting of some fields. All keyword arguments passed in will be used as parameters for all the fields in the `fields` parameter. Example:

```
Search().highlight('title', 'body', fragment_size=50)
```

will produce the equivalent of:

```
{
    "highlight": {
        "fields": {
            "body": {"fragment_size": 50},
            "title": {"fragment_size": 50}
        }
    }
}
```

If you want to have different options for different fields you can call `highlight` twice:

```
Search().highlight('title', fragment_size=50).highlight('body', fragment_
↪size=100)
```

which will produce:

```
{
    "highlight": {
```

```
        "fields": {
            "body": {"fragment_size": 100},
            "title": {"fragment_size": 50}
        }
    }
}
```

highlight_options(***kwargs*)
Update the global highlighting options used for this request. For example:

```
s = Search()
s = s.highlight_options(order='score')
```

response_class(*cls*)
Override the default wrapper used for the response.

scan()
Turn the search into a scan search and return a generator that will iterate over all the documents matching the query.

Use `params` method to specify any additional arguments you with to pass to the underlying `scan` helper from `elasticsearch-py` - https://elasticsearch-py.readthedocs.io/en/master/helpers.html#elasticsearch.helpers.scan

script_fields(***kwargs*)
Define script fields to be calculated on hits. See https://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-script-fields.html for more details.

Example:

```
s = Search()
s = s.script_fields(times_two="doc['field'].value * 2")
s = s.script_fields(
    times_three={
        'script': {
            'inline': "doc['field'].value * params.n",
            'params': {'n': 3}
        }
    }
)
```

sort(**keys*)
Add sorting information to the search request. If called without arguments it will remove all sort requirements. Otherwise it will replace them. Acceptable arguments are:

```
'some.field'
'-some.other.field'
{'different.field': {'any': 'dict'}}
```

so for example:

```
s = Search().sort(
    'category',
    '-title',
    {"price" : {"order" : "asc", "mode" : "avg"}}
)
```

will sort by `category`, `title` (in descending order) and `price` in ascending order using the `avg` mode.

The API returns a copy of the Search object and can thus be chained.

source(*fields=None*, ***kwargs*)
Selectively control how the _source field is returned.

Parameters **source** – wildcard string, array of wildcards, or dictionary of includes and excludes

If `source` is None, the entire document will be returned for each hit. If source is a dictionary with keys of 'include' and/or 'exclude' the fields will be either included or excluded appropriately.

Calling this multiple times with the same named parameter will override the previous values with the new ones.

Example:

```
s = Search()
s = s.source(include=['obj1.*'], exclude=["*.description"])

s = Search()
s = s.source(include=['obj1.*']).source(exclude=["*.description"])
```

suggest(*name*, *text*, ***kwargs*)
Add a suggestions request to the search.

Parameters

- **name** – name of the suggestion
- **text** – text to suggest on

All keyword arguments will be added to the suggestions body. For example:

```
s = Search()
s = s.suggest('suggestion-1', 'Elasticsearch', term={'field': 'body'})
```

to_dict(*count=False*, ***kwargs*)
Serialize the search into the dictionary that will be sent over as the request's body.

Parameters **count** – a flag to specify we are interested in a body for count - no aggregations, no pagination bounds etc.

All additional keyword arguments will be included into the dictionary.

update_from_dict(*d*)
Apply options from a serialized body to the current instance. Modifies the object in-place. Used mostly by `from_dict`.

class elasticsearch_dsl.**MultiSearch**(***kwargs*)
Combine multiple *Search* objects into a single request.

add(*search*)
Adds a new *Search* object to the request:

```
ms = MultiSearch(index='my-index')
ms = ms.add(Search(doc_type=Category).filter('term', category='python'))
ms = ms.add(Search(doc_type=Blog))
```

execute(*ignore_cache=False*, *raise_on_error=True*)
Execute the multi search request and return a list of search results.

7.5.2 Document

class elasticsearch_dsl.**DocType**(*meta=None*, ***kwargs*)
Model-like class for persisting documents in elasticsearch.

delete(*using=None*, *index=None*, ***kwargs*)
Delete the instance in elasticsearch.

Parameters

- **index** – elasticsearch index to use, if the DocType is associated with an index this can be omitted.
- **using** – connection alias to use, defaults to 'default'

Any additional keyword arguments will be passed to Elasticsearch.delete unchanged.

classmethod get(*id*, *using=None*, *index=None*, ***kwargs*)
Retrieve a single document from elasticsearch using it's id.

Parameters

- **id** – id of the document to be retireved
- **index** – elasticsearch index to use, if the DocType is associated with an index this can be omitted.
- **using** – connection alias to use, defaults to 'default'

Any additional keyword arguments will be passed to Elasticsearch.get unchanged.

classmethod init(*index=None*, *using=None*)
Create the index and populate the mappings in elasticsearch.

classmethod mget(*docs*, *using=None*, *index=None*, *raise_on_error=True*, *missing='none'*, ***kwargs*)
Retrieve multiple document by their ids. Returns a list of instances in the same order as requested.

Parameters

- **docs** – list of ids of the documents to be retireved or a list of document specifications as per https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-multi-get.html
- **index** – elasticsearch index to use, if the DocType is associated with an index this can be omitted.
- **using** – connection alias to use, defaults to 'default'
- **missing** – what to do when one of the documents requested is not found. Valid options are 'none' (use None), 'raise' (raise NotFoundError) or 'skip' (ignore the missing document).

Any additional keyword arguments will be passed to Elasticsearch.mget unchanged.

save(*using=None*, *index=None*, *validate=True*, ***kwargs*)
Save the document into elasticsearch. If the document doesn't exist it is created, it is overwritten otherwise. Returns True if this operations resulted in new document being created.

Parameters

- **index** – elasticsearch index to use, if the DocType is associated with an index this can be omitted.
- **using** – connection alias to use, defaults to 'default'
- **validate** – set to False to skip validating the document

Any additional keyword arguments will be passed to `Elasticsearch.index` unchanged.

classmethod search(*using=None*, *index=None*)
Create an *Search* instance that will search over this `DocType`.

to_dict(*include_meta=False*)
Serialize the instance into a dictionary so that it can be saved in elasticsearch.

Parameters **include_meta** – if set to `True` will include all the metadata (`_index`, `_type`, `_id` etc). Otherwise just the document's data is serialized. This is useful when passing multiple instances into `elasticsearch.helpers.bulk`.

update(*using=None*, *index=None*, *detect_noop=True*, *doc_as_upsert=False*, ***fields*)
Partial update of the document, specify fields you wish to update and both the instance and the document in elasticsearch will be updated:

```
doc = MyDocument(title='Document Title!')
doc.save()
doc.update(title='New Document Title!')
```

Parameters

- **index** – elasticsearch index to use, if the `DocType` is associated with an index this can be omitted.
- **using** – connection alias to use, defaults to `'default'`

Any additional keyword arguments will be passed to `Elasticsearch.update` unchanged.

7.5.3 Index

class elasticsearch_dsl.**Index**(*name*, *using='default'*)

Parameters

- **name** – name of the index
- **using** – connection alias to use, defaults to `'default'`

analyze(***kwargs*)
Perform the analysis process on a text and return the tokens breakdown of the text.

Any additional keyword arguments will be passed to `Elasticsearch.indices.analyze` unchanged.

clear_cache(***kwargs*)
Clear all caches or specific cached associated with the index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.clear_cache` unchanged.

close(***kwargs*)
Closes the index in elasticsearch.

Any additional keyword arguments will be passed to `Elasticsearch.indices.close` unchanged.

create(***kwargs*)
Creates the index in elasticsearch.

Any additional keyword arguments will be passed to `Elasticsearch.indices.create` unchanged.

delete(***kwargs*)
Deletes the index in elasticsearch.

Any additional keyword arguments will be passed to `Elasticsearch.indices.delete` unchanged.

delete_alias(***kwargs*)
Delete specific alias.

Any additional keyword arguments will be passed to `Elasticsearch.indices.delete_alias` unchanged.

exists(***kwargs*)
Returns `True` if the index already exists in elasticsearch.

Any additional keyword arguments will be passed to `Elasticsearch.indices.exists` unchanged.

exists_alias(***kwargs*)
Return a boolean indicating whether given alias exists for this index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.exists_alias` unchanged.

exists_type(***kwargs*)
Check if a type/types exists in the index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.exists_type` unchanged.

flush(***kwargs*)
Preforms a flush operation on the index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.flush` unchanged.

flush_synced(***kwargs*)
Perform a normal flush, then add a generated unique marker (sync_id) to all shards.

Any additional keyword arguments will be passed to `Elasticsearch.indices.flush_synced` unchanged.

forcemerge(***kwargs*)
The force merge API allows to force merging of the index through an API. The merge relates to the number of segments a Lucene index holds within each shard. The force merge operation allows to reduce the number of segments by merging them.

This call will block until the merge is complete. If the http connection is lost, the request will continue in the background, and any new requests will block until the previous force merge is complete.

Any additional keyword arguments will be passed to `Elasticsearch.indices.forcemerge` unchanged.

get(***kwargs*)
The get index API allows to retrieve information about the index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.get` unchanged.

get_alias(***kwargs*)
Retrieve a specified alias.

Any additional keyword arguments will be passed to `Elasticsearch.indices.get_alias` unchanged.

get_field_mapping(***kwargs*)
Retrieve mapping definition of a specific field.

Any additional keyword arguments will be passed to `Elasticsearch.indices.get_field_mapping` unchanged.

get_mapping(***kwargs*)
Retrieve specific mapping definition for a specific type.

Any additional keyword arguments will be passed to `Elasticsearch.indices.get_mapping` unchanged.

get_settings(***kwargs*)
Retrieve settings for the index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.get_settings` unchanged.

get_upgrade(***kwargs*)
Monitor how much of the index is upgraded.

Any additional keyword arguments will be passed to `Elasticsearch.indices.get_upgrade` unchanged.

open(***kwargs*)
Opens the index in elasticsearch.

Any additional keyword arguments will be passed to `Elasticsearch.indices.open` unchanged.

put_alias(***kwargs*)
Create an alias for the index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.put_alias` unchanged.

put_mapping(***kwargs*)
Register specific mapping definition for a specific type.

Any additional keyword arguments will be passed to `Elasticsearch.indices.put_mapping` unchanged.

put_settings(***kwargs*)
Change specific index level settings in real time.

Any additional keyword arguments will be passed to `Elasticsearch.indices.put_settings` unchanged.

recovery(***kwargs*)
The indices recovery API provides insight into on-going shard recoveries for the index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.recovery` unchanged.

refresh(***kwargs*)
Preforms a refresh operation on the index.

Any additional keyword arguments will be passed to `Elasticsearch.indices.refresh` unchanged.

save()
Sync the index definition with elasticsearch, creating the index if it doesn't exist and updating its settings and mappings if it does.

Note some settings and mapping changes cannot be done on an open index (or at all on an existing index) and for those this method will fail with the underlying exception.

search()
Return a *Search* object searching over this index and its DocTypes.

segments(***kwargs*)
Provide low level segments information that a Lucene index (shard level) is built with.

Any additional keyword arguments will be passed to Elasticsearch.indices.segments unchanged.

shard_stores(***kwargs*)
Provides store information for shard copies of the index. Store information reports on which nodes shard copies exist, the shard copy version, indicating how recent they are, and any exceptions encountered while opening the shard index or from earlier engine failure.

Any additional keyword arguments will be passed to Elasticsearch.indices.shard_stores unchanged.

shrink(***kwargs*)
The shrink index API allows you to shrink an existing index into a new index with fewer primary shards. The number of primary shards in the target index must be a factor of the shards in the source index. For example an index with 8 primary shards can be shrunk into 4, 2 or 1 primary shards or an index with 15 primary shards can be shrunk into 5, 3 or 1. If the number of shards in the index is a prime number it can only be shrunk into a single primary shard. Before shrinking, a (primary or replica) copy of every shard in the index must be present on the same node.

Any additional keyword arguments will be passed to Elasticsearch.indices.shrink unchanged.

stats(***kwargs*)
Retrieve statistics on different operations happening on the index.

Any additional keyword arguments will be passed to Elasticsearch.indices.stats unchanged.

upgrade(***kwargs*)
Upgrade the index to the latest format.

Any additional keyword arguments will be passed to Elasticsearch.indices.upgrade unchanged.

validate_query(***kwargs*)
Validate a potentially expensive query without executing it.

Any additional keyword arguments will be passed to Elasticsearch.indices.validate_query unchanged.

7.5.4 Faceted Search

class elasticsearch_dsl.**FacetedSearch**(*query=None*, *filters={}*, *sort=()*)
Abstraction for creating faceted navigation searches that takes care of composing the queries, aggregations and filters as needed as well as presenting the results in an easy-to-consume fashion:

```
class BlogSearch(FacetedSearch):
    index = 'blogs'
    doc_types = [Blog, Post]
    fields = ['title^5', 'category', 'description', 'body']

    facets = {
```

```
        'type': TermsFacet(field='_type'),
        'category': TermsFacet(field='category'),
        'weekly_posts': DateHistogramFacet(field='published_from', interval='week
↪')
    }

    def search(self):
        ' Override search to add your own filters '
        s = super(BlogSearch, self).search()
        return s.filter('term', published=True)

# when using:
blog_search = BlogSearch("web framework", filters={"category": "python"})

# supports pagination
blog_search[10:20]

response = blog_search.execute()

# easy access to aggregation results:
for category, hit_count, is_selected in response.facets.category:
    print(
        "Category %s has %d hits%s." % (
            category,
            hit_count,
            ' and is chosen' if is_selected else ''
        )
    )
```

Parameters

- **query** – the text to search for
- **filters** – facet values to filter
- **sort** – sort information to be passed to *Search*

add_filter(*name*, *filter_values*)
Add a filter for a facet.

aggregate(*search*)
Add aggregations representing the facets selected, including potential filters.

build_search()
Construct the `Search` object.

execute()
Execute the search and return the response.

filter(*search*)
Add a `post_filter` to the search request narrowing the results based on the facet filters.

highlight(*search*)
Add highlighting for all the fields

query(*search*, *query*)
Add query part to `search`.

Override this if you wish to customize the query used.

search()
Construct the Search object.

sort(*search*)
Add sorting information to the request.

7.6 Contribution Guide

If you have a bugfix or new feature that you would like to contribute to elasticsearch-dsl-py, please find or open an issue about it first. Talk about what you would like to do. It may be that somebody is already working on it, or that there are particular issues that you should know about before implementing the change.

We enjoy working with contributors to get their code accepted. There are many approaches to fixing a problem and it is important to find the best approach before writing too much code.

The process for contributing to any of the Elasticsearch repositories is similar.

1. Please make sure you have signed the Contributor License Agreement. We are not asking you to assign copyright to us, but to give us the right to distribute your code without restriction. We ask this of all contributors in order to assure our users of the origin and continuing existence of the code. You only need to sign the CLA once.
2. Run the test suite to ensure your changes do not break existing code:

```
$ python setup.py test
```

3. Rebase your changes. Update your local repository with the most recent code from the main elasticsearch-dsl-py repository, and rebase your branch on top of the latest master branch. We prefer your changes to be squashed into a single commit.
4. Submit a pull request. Push your local changes to your forked copy of the repository and submit a pull request. In the pull request, describe what your changes do and mention the number of the issue where discussion has taken place, eg "Closes #123. Please consider adding or modifying tests related to your changes.

Then sit back and wait. There will probably be discussion about the pull request and, if any changes are needed, we would love to work with you to get your pull request merged into elasticsearch-dsl-py.

7.7 Changelog

7.7.1 6.1.0 (2018-01-09)

- Removed `String` field.
- Fixed issue with `Object`/`Nested` deserialization

7.7.2 6.0.1 (2018-01-02)

Fixing wheel package for Python 2.7 (#803)

7.7.3 6.0.0 (2018-01-01)

Backwards incompatible release compatible with elasticsearch 6.0, changes include:

- use `doc` as default `DocType` name, this change includes: * `DocType._doc_type.matches` method is now used to determine which `DocType` should be used for a hit instead of just checking `_type`
- `Nested` and `Object` field refactoring using newly introduced `InnerDoc` class. To define a `Nested`/`Object` field just define the `InnerDoc` subclass and then use it when defining the field:

```
class Comment(InnerDoc):
    body = Text()
    created_at = Date()

class Blog(DocType):
    comments = Nested(Comment)
```

- methods on `connections` singleton are now exposed on the `connections` module directly.
- field values are now only deserialized when coming from elasticsearch (via `from_es` method) and not when assigning values in python (either by direct assignment or in `__init__`).

7.7.4 5.4.0 (2017-12-06)

- fix `ip_range` aggregation and rename the class to `IPRange`. `Iprange` is kept for bw compatibility
- fix bug in loading an aggregation with meta data from dict
- add support for `normalizer` parameter of `Keyword` fields
- `IndexTemplate` can now be specified using the same API as `Index`
- `Boolean` field now accepts `"false"` as `False`

7.7.5 5.3.0 (2017-05-18)

- fix constant score query definition
- `DateHistogramFacet` now works with `datetime` objects
- respect ___ in field names when creating queries from dict

7.7.6 5.2.0 (2017-03-26)

- make sure all response structers are pickleable (for caching)
- adding `exclude` to `Search`
- fix metric aggregation deserialization
- expose all index-level APIs on `Index` class
- adding `delete` to `Search` which calls `delete_by_query` API

7.7.7 5.1.0 (2017-01-08)

- Renamed `Result` and `ResultMeta` to `Hit` and `HitMeta` respectively
- `Response` now stores `Search` which it gets as first arg to `__init__`
- aggregation results are now wrapped in classes and properly deserialized
- `Date` fields now allow for numerical timestamps in the java format (in millis)

- Added API documentation
- replaced generated classes with manually created

7.7.8 5.0.0 (2016-11-04)

Version compatible with elasticsearch 5.0.

Breaking changes:

- `String` field type has been deprecated in favor of `Text` and `Keyword`
- `fields` method has been removed in favor of `source` filtering

7.7.9 2.2.0 (2016-11-04)

- accessing missing string fields no longer returnd `''` but returns `None` instead.
- fix issues with bool's `|` and `&` operators and `minimum_should_match`

7.7.10 2.1.0 (2016-06-29)

- `inner_hits` are now also wrapped in `Response`
- `+` operator is deprecated, `.query()` now uses `&` to combine queries
- added `mget` method to `DocType`
- fixed validation for "empty" values like `''` and `[]`

7.7.11 2.0.0 (2016-02-18)

Compatibility with Elasticsearch 2.x:

- Filters have been removed and additional queries have been added. Instead of `F` objects you can now use `Q`.
- `Search.filter` is now just a shortcut to add queries in filter context
- support for pipeline aggregations added

Backwards incompatible changes:

- list of analysis objects and classes was removed, any string used as tokenizer, char or token filter or analyzer will be treated as a builtin
- internal method `Field.to_python` has been renamed to `deserialize` and an optional serialization mechanic for fields has been added.
- Custom response class is now set by `response_class` method instead of a kwarg to `Search.execute`

Other changes:

- `FacetedSearch` now supports pagination via slicing

7.7.12 0.0.10 (2016-01-24)

- `Search` can now be iterated over to get back hits
- `Search` now caches responses from Elasticsearch
- `DateHistogramFacet` now defaults to returning empty intervals
- `Search` no longer accepts positional parameters
- Experimental `MultiSearch` API
- added option to talk to `_suggest` endpoint (`execute_suggest`)

7.7.13 0.0.9 (2015-10-26)

- `FacetedSearch` now uses its own `Facet` class instead of built in aggregations

7.7.14 0.0.8 (2015-08-28)

- `0.0.5` and `0.0.6` was released with broken .tar.gz on pypi, just a build fix

7.7.15 0.0.5 (2015-08-27)

- added support for (index/search)_analyzer via #143, thanks @wkiser!
- even keys accessed via `['field']` on `AttrDict` will be wrapped in `Attr[Dict|List]` for consistency
- Added a convenient option to specify a custom `doc_class` to wrap inner/Nested documents
- `blank` option has been removed
- `AttributeError` is no longer raised when accessing an empty field.
- added `required` flag to fields and validation hooks to fields and (sub)documents
- removed `get` method from `AttrDict`. Use `getattr(d, key, default)` instead.
- added `FacetedSearch` for easy declarative faceted navigation

7.7.16 0.0.4 (2015-04-24)

- Metadata fields (such as id, parent, index, version etc) must be stored (and retrieved) using the `meta` attribute (#58) on both `Result` and `DocType` objects or using their underscored variants (`_id`, `_parent` etc)
- query on Search can now be directly assigned
- `suggest` method added to `Search`
- `Search.doc_type` now accepts `DocType` subclasses directly
- `Properties.property` method renamed to `field` for consistency
- Date field now raises `ValidationException` on incorrect data

7.7.17 0.0.3 (2015-01-23)

Added persistence layer (`Mapping` and `DocType`), various fixes and improvements.

7.7.18 0.0.2 (2014-08-27)

Fix for python 2

7.7.19 0.0.1 (2014-08-27)

Initial release.

Python Module Index

e

elasticsearch_dsl, 32

Index

A

add() (elasticsearch_dsl.MultiSearch method), 35
add_filter() (elasticsearch_dsl.FacetedSearch method), 41
aggregate() (elasticsearch_dsl.FacetedSearch method), 41
analyze() (elasticsearch_dsl.Index method), 37

B

build_search() (elasticsearch_dsl.FacetedSearch method), 41

C

clear_cache() (elasticsearch_dsl.Index method), 37
close() (elasticsearch_dsl.Index method), 37
count() (elasticsearch_dsl.Search method), 33
create() (elasticsearch_dsl.Index method), 37

D

delete() (elasticsearch_dsl.DocType method), 36
delete() (elasticsearch_dsl.Index method), 37
delete() (elasticsearch_dsl.Search method), 33
delete_alias() (elasticsearch_dsl.Index method), 38
DocType (class in elasticsearch_dsl), 36

E

elasticsearch_dsl (module), 32
execute() (elasticsearch_dsl.FacetedSearch method), 41
execute() (elasticsearch_dsl.MultiSearch method), 35
execute() (elasticsearch_dsl.Search method), 33
exists() (elasticsearch_dsl.Index method), 38
exists_alias() (elasticsearch_dsl.Index method), 38
exists_type() (elasticsearch_dsl.Index method), 38

F

FacetedSearch (class in elasticsearch_dsl), 40
filter() (elasticsearch_dsl.FacetedSearch method), 41
flush() (elasticsearch_dsl.Index method), 38
flush_synced() (elasticsearch_dsl.Index method), 38
forcemerge() (elasticsearch_dsl.Index method), 38
from_dict() (elasticsearch_dsl.Search class method), 33

G

get() (elasticsearch_dsl.DocType class method), 36
get() (elasticsearch_dsl.Index method), 38
get_alias() (elasticsearch_dsl.Index method), 38
get_field_mapping() (elasticsearch_dsl.Index method), 38
get_mapping() (elasticsearch_dsl.Index method), 39
get_settings() (elasticsearch_dsl.Index method), 39
get_upgrade() (elasticsearch_dsl.Index method), 39

H

highlight() (elasticsearch_dsl.FacetedSearch method), 41
highlight() (elasticsearch_dsl.Search method), 33
highlight_options() (elasticsearch_dsl.Search method), 34

I

Index (class in elasticsearch_dsl), 37
init() (elasticsearch_dsl.DocType class method), 36

M

mget() (elasticsearch_dsl.DocType class method), 36
MultiSearch (class in elasticsearch_dsl), 35

O

open() (elasticsearch_dsl.Index method), 39

P

put_alias() (elasticsearch_dsl.Index method), 39
put_mapping() (elasticsearch_dsl.Index method), 39
put_settings() (elasticsearch_dsl.Index method), 39

Q

query() (elasticsearch_dsl.FacetedSearch method), 41

R

recovery() (elasticsearch_dsl.Index method), 39
refresh() (elasticsearch_dsl.Index method), 39
response_class() (elasticsearch_dsl.Search method), 34

S

save() (elasticsearch_dsl.DocType method), 36
save() (elasticsearch_dsl.Index method), 39
scan() (elasticsearch_dsl.Search method), 34
script_fields() (elasticsearch_dsl.Search method), 34
Search (class in elasticsearch_dsl), 32
search() (elasticsearch_dsl.DocType class method), 37
search() (elasticsearch_dsl.FacetedSearch method), 41
search() (elasticsearch_dsl.Index method), 40
segments() (elasticsearch_dsl.Index method), 40
shard_stores() (elasticsearch_dsl.Index method), 40
shrink() (elasticsearch_dsl.Index method), 40
sort() (elasticsearch_dsl.FacetedSearch method), 42
sort() (elasticsearch_dsl.Search method), 34
source() (elasticsearch_dsl.Search method), 35
stats() (elasticsearch_dsl.Index method), 40
suggest() (elasticsearch_dsl.Search method), 35

T

to_dict() (elasticsearch_dsl.DocType method), 37
to_dict() (elasticsearch_dsl.Search method), 35

U

update() (elasticsearch_dsl.DocType method), 37
update_from_dict() (elasticsearch_dsl.Search method), 35
upgrade() (elasticsearch_dsl.Index method), 40

V

validate_query() (elasticsearch_dsl.Index method), 40