Django Elasticsearch DSL Documentation

Release 7.1.1

sabricot and others

Oct 01, 2023

Contents

Contents:

CHAPTER 1

Django Elasticsearch DSL

Django Elasticsearch DSL is a package that allows indexing of django models in elasticsearch. It is built as a thin wrapper around elasticsearch-dsl-py so you can use all the features developed by the elasticsearch-dsl-py team.

You can view the full documentation at https://django-elasticsearch-dsl.readthedocs.io

1.1 Features

- Based on elasticsearch-dsl-py so you can make queries with the Search class.
- Django signal receivers on save and delete for keeping Elasticsearch in sync.
- Management commands for creating, deleting, rebuilding and populating indices.
- Elasticsearch auto mapping from django models fields.
- Complex field type support (ObjectField, NestedField).
- Index fast using *parallel* indexing.
- Requirements
  - Django >= 3.2
  - Python 3.8, 3.9, 3.10, 3.11

Elasticsearch Compatibility: The library is compatible with all Elasticsearch versions since 5.x **but you have to use a matching major version:**

- For Elasticsearch 8.0 and later, use the major version 8 (8.x.y) of the library.

- For Elasticsearch 7.0 and later, use the major version 7 (7.x.y) of the library.
- For Elasticsearch 6.0 and later, use the major version 6 (6.x.y) of the library.

```
# Elasticsearch 8.x
elasticsearch-dsl>=8.0.0,<9.0.0

# Elasticsearch 7.x
elasticsearch-dsl>=7.0.0,<8.0.0

# Elasticsearch 6.x
elasticsearch-dsl>=6.0.0,<7.0.0
```

CHAPTER 2

Quickstart

2.1 Install and configure

Install Django Elasticsearch DSL:

```
pip install django-elasticsearch-dsl
```

Then add `django_elasticsearch_dsl` to the INSTALLED_APPS

You must define `ELASTICSEARCH_DSL` in your django settings.

For example:

```
ELASTICSEARCH_DSL={
    'default': {
        'hosts': 'localhost:9200',
        'http_auth': ('username', 'password')
    }
}
```

`ELASTICSEARCH_DSL` is then passed to `elasticsearch-dsl-py.connections.configure` (see here).

2.2 Declare data to index

Then for a model:

```
# models.py

class Car(models.Model):
    name = models.CharField()
    color = models.CharField()
    description = models.TextField()
```

(continues on next page)

(continued from previous page)

```
    type = models.IntegerField(choices=[
        (1, "Sedan"),
        (2, "Truck"),
        (4, "SUV"),
    ])
```

To make this model work with Elasticsearch, create a subclass of `django_elasticsearch_dsl.Document`, create a `class Index` inside the `Document` class to define your Elasticsearch indices, names, settings etc and at last register the class using `registry.register_document` decorator. It is required to define `Document` class in `documents.py` in your app directory.

```
# documents.py

from django_elasticsearch_dsl import Document
from django_elasticsearch_dsl.registries import registry
from .models import Car


@registry.register_document
class CarDocument(Document):
    class Index:
        # Name of the Elasticsearch index
        name = 'cars'
        # See Elasticsearch Indices API reference for available settings
        settings = {'number_of_shards': 1,
                    'number_of_replicas': 0}

    class Django:
        model = Car # The model associated with this Document

        # The fields of the model you want to be indexed in Elasticsearch
        fields = [
            'name',
            'color',
            'description',
            'type',
        ]

        # Ignore auto updating of Elasticsearch when a model is saved
        # or deleted:
        # ignore_signals = True

        # Configure how the index should be refreshed after an update.
        # See Elasticsearch documentation for supported options:
        # https://www.elastic.co/guide/en/elasticsearch/reference/master/docs-refresh.
↪html
        # This per-Document setting overrides settings.ELASTICSEARCH_DSL_AUTO_REFRESH.
        # auto_refresh = False

        # Paginate the django queryset used to populate the index with the specified␣
↪size
        # (by default it uses the database driver's default setting)
        # queryset_pagination = 5000
```

2.3 Populate

To create and populate the Elasticsearch index and mapping use the search_index command:

```
$ ./manage.py search_index --rebuild
```

Now, when you do something like:

```
car = Car(
    name="Car one",
    color="red",
    type=1,
    description="A beautiful car"
)
car.save()
```

The object will be saved in Elasticsearch too (using a signal handler).

2.4 Search

To get an elasticsearch-dsl-py Search instance, use:

```
s = CarDocument.search().filter("term", color="red")

# or

s = CarDocument.search().query("match", description="beautiful")

for hit in s:
    print(
        "Car name : {}, description {}".format(hit.name, hit.description)
    )
```

The previous example returns a result specific to elasticsearch_dsl, but it is also possible to convert the elastisearch result into a real django queryset, just be aware that this costs a sql request to retrieve the model instances with the ids returned by the elastisearch query.

```
s = CarDocument.search().filter("term", color="blue")[:30]
qs = s.to_queryset()
# qs is just a django queryset and it is called with order_by to keep
# the same order as the elasticsearch result.
for car in qs:
    print(car.name)
```

CHAPTER 3

Index

In typical scenario using *class Index* on a *Document* class is sufficient to perform any action. In a few cases though it can be useful to manipulate an Index object directly.

To define an Elasticsearch index you must instantiate a `elasticsearch_dsl.Index` class and set the name and settings of the index. After you instantiate your class, you need to associate it with the Document you want to put in this Elasticsearch index and also add the *registry.register_document* decorator.

```
# documents.py
from elasticsearch_dsl import Index
from django_elasticsearch_dsl import Document
from .models import Car, Manufacturer

# The name of your index
car = Index('cars')
# See Elasticsearch Indices API reference for available settings
car.settings(
    number_of_shards=1,
    number_of_replicas=0
)

@registry.register_document
@car.document
class CarDocument(Document):
    class Django:
        model = Car
        fields = [
            'name',
            'color',
        ]

@registry.register_document
class ManufacturerDocument(Document):
    class Index:
        name = 'manufacture'
```

(continues on next page)

(continued from previous page)

```
        settings = {'number_of_shards': 1,
                    'number_of_replicas': 0}

    class Django:
        model = Manufacturer
        fields = [
            'name',
            'country_code',
        ]
```

When you execute the command:

```
$ ./manage.py search_index --rebuild
```

This will create two index named `cars` and `manufacture` in Elasticsearch with appropriate mapping.

** If your model have huge amount of data, its preferred to use *parallel* indexing. To do that, you can pass *–parallel* flag while reindexing or populating. **

3.1 Signals

- **`django_elasticsearch_dsl.signals.post_index`** Sent after document indexing is completed. (not applicable for `parallel` indexing). Provides the following arguments:

  `sender` A subclass of `django_elasticsearch_dsl.documents.DocType` used to perform indexing.

  `instance` A `django_elasticsearch_dsl.documents.DocType` subclass instance.

  `actions` A generator containing document data that were sent to elasticsearch for indexing.

  `response` The response from `bulk()` function of `elasticsearch-py`, which includes `success` count and `failed` count or `error` list.

CHAPTER 4

Fields

Once again the `django_elasticsearch_dsl.fields` are subclasses of elasticsearch-dsl-py fields. They just add support for retrieving data from django models.

4.1 Using Different Attributes for Model Fields

Let's say you don't want to store the type of the car as an integer, but as the corresponding string instead. You need some way to convert the type field on the model to a string, so we'll just add a method for it:

```
# models.py

class Car(models.Model):
    # ... #
    def type_to_string(self):
        """Convert the type field to its string representation
        (the boneheaded way).
        """
        if self.type == 1:
            return "Sedan"
        elif self.type == 2:
            return "Truck"
        else:
            return "SUV"
```

Now we need to tell our `Document` subclass to use that method instead of just accessing the `type` field on the model directly. Change the CarDocument to look like this:

```
# documents.py

from django_elasticsearch_dsl import Document, fields

# ... #
```

(continues on next page)

(continued from previous page)

```
@registry.register_document
class CarDocument(Document):
    # add a string field to the Elasticsearch mapping called type, the
    # value of which is derived from the model's type_to_string attribute
    type = fields.TextField(attr="type_to_string")

    class Django:
        model = Car
        # we removed the type field from here
        fields = [
            'name',
            'color',
            'description',
        ]
```

After a change like this we need to rebuild the index with:

```
$ ./manage.py search_index --rebuild
```

4.2 Using prepare_field

Sometimes, you need to do some extra prepping before a field should be saved to Elasticsearch. You can add a `prepare_foo(self, instance)` method to a Document (where foo is the name of the field), and that will be called when the field needs to be saved.

```
# documents.py

# ... #

class CarDocument(Document):
    # ... #

    foo = TextField()

    def prepare_foo(self, instance):
        return " ".join(instance.foos)
```

4.3 Handle relationship with NestedField/ObjectField

For example for a model with ForeignKey relationships.

```
# models.py

class Car(models.Model):
    name = models.CharField()
    color = models.CharField()
    manufacturer = models.ForeignKey('Manufacturer')

class Manufacturer(models.Model):
    name = models.CharField()
    country_code = models.CharField(max_length=2)
```

(continues on next page)

(continued from previous page)

```
    created = models.DateField()

class Ad(models.Model):
    title = models.CharField()
    description = models.TextField()
    created = models.DateField(auto_now_add=True)
    modified = models.DateField(auto_now=True)
    url = models.URLField()
    car = models.ForeignKey('Car', related_name='ads')
```

You can use an ObjectField or a NestedField.

```
# documents.py

from django_elasticsearch_dsl import Document, fields
from .models import Car, Manufacturer, Ad

@registry.register_document
class CarDocument(Document):
    manufacturer = fields.ObjectField(properties={
        'name': fields.TextField(),
        'country_code': fields.TextField(),
    })
    ads = fields.NestedField(properties={
        'description': fields.TextField(analyzer=html_strip),
        'title': fields.TextField(),
        'pk': fields.IntegerField(),
    })

    class Index:
        name = 'cars'

    class Django:
        model = Car
        fields = [
            'name',
            'color',
        ]
        related_models = [Manufacturer, Ad]  # Optional: to ensure the Car will be re-
↪saved when Manufacturer or Ad is updated

    def get_queryset(self):
        """Not mandatory but to improve performance we can select related in one sql␣
↪request"""
        return super(CarDocument, self).get_queryset().select_related(
            'manufacturer'
        )

    def get_instances_from_related(self, related_instance):
        """If related_models is set, define how to retrieve the Car instance(s) from␣
↪the related model.
        The related_models option should be used with caution because it can lead in␣
↪the index
        to the updating of a lot of items.
        """
        if isinstance(related_instance, Manufacturer):
            return related_instance.car_set.all()
```

(continues on next page)

(continued from previous page)

```
        elif isinstance(related_instance, Ad):
            return related_instance.car
```

4.4 Field Classes

Most Elasticsearch field types are supported. The `attr` argument is a dotted "attribute path" which will be looked up on the model using Django template semantics (dict lookup, attribute lookup, list index lookup). By default the attr argument is set to the field name.

For the rest, the field properties are the same as elasticsearch-dsl fields.

So for example you can use a custom analyzer:

```
# documents.py

# ... #

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

@registry.register_document
class CarDocument(Document):
    description = fields.TextField(
        analyzer=html_strip,
        fields={'raw': fields.KeywordField()}
    )

    class Django:
        model = Car
        fields = [
            'name',
            'color',
        ]
```

4.5 Available Fields

- Simple Fields
  - `BooleanField(attr=None, **elasticsearch_properties)`
  - `ByteField(attr=None, **elasticsearch_properties)`
  - `CompletionField(attr=None, **elasticsearch_properties)`
  - `DateField(attr=None, **elasticsearch_properties)`
  - `DoubleField(attr=None, **elasticsearch_properties)`
  - `FileField(attr=None, **elasticsearch_properties)`
  - `FloatField(attr=None, **elasticsearch_properties)`

  - `IntegerField(attr=None, **elasticsearch_properties)`
  - `IpField(attr=None, **elasticsearch_properties)`
  - `KeywordField(attr=None, **elasticsearch_properties)`
  - `GeoPointField(attr=None, **elasticsearch_properties)`
  - `GeoShapeField(attr=None, **elasticsearch_properties)`
  - `ShortField(attr=None, **elasticsearch_properties)`
  - `TextField(attr=None, **elasticsearch_properties)`
- Complex Fields
  - `ObjectField(properties, attr=None, **elasticsearch_properties)`
  - `NestedField(properties, attr=None, **elasticsearch_properties)`

`properties` is a dict where the key is a field name, and the value is a field instance.

4.6 Field Mapping

Django Elasticsearch DSL maps most of the django fields appropriate Elasticsearch Field. You can find the field mapping on *documents.py* file in the *model_field_class_to_field_class* variable. If you need to change the behavior of this mapping, or add mapping for your custom field, you can do so by overwriting the classmethod *get_model_field_class_to_field_class*. Remember, you need to inherit *django_elasticsearch_dsl.fields.DEDField* for your custom field. Like following

```
from django_elasticsearch_dsl.fields import DEDField

class MyCustomDEDField(DEDField, ElasticsearchField):
    pass

@classmethod
def get_model_field_class_to_field_class(cls):
    field_mapping = super().get_model_field_class_to_field_class()
    field_mapping[MyCustomDjangoField] = MyCustomDEDField
```

4.7 Document id

The elasticsearch document id (`_id`) is not strictly speaking a field, as it is not part of the document itself. The default behavior of `django_elasticsearch_dsl` is to use the primary key of the model as the document's id (`pk` or `id`). Nevertheless, it can sometimes be useful to change this default behavior. For this, one can redefine the `generate_id(cls, instance)` class method of the `Document` class.

For example, to use an article's slug as the elasticsearch `_id` instead of the article's integer id, one could use:

```
# models.py

from django.db import models

class Article(models.Model):
    # ... #

    slug = models.SlugField(
```

(continues on next page)

(continued from previous page)

```
        max_length=255,
        unique=True,
    )

    # ... #


# documents.py

from .models import Article

class ArticleDocument(Document):
    class Django:
        model = Article

    # ... #

    @classmethod
    def generate_id(cls, article):
        return article.slug
```

CHAPTER 5

Settings

5.1 ELASTICSEARCH_DSL_AUTOSYNC

Default: `True`

Set to `False` to globally disable auto-syncing.

5.2 ELASTICSEARCH_DSL_INDEX_SETTINGS

Default: `{}`

Additional options passed to the elasticsearch-dsl Index settings (like `number_of_replicas` or `number_of_shards`).

5.3 ELASTICSEARCH_DSL_AUTO_REFRESH

Default: `True`

Set to `False` not force an index refresh with every save.

5.4 ELASTICSEARCH_DSL_SIGNAL_PROCESSOR

This (optional) setting controls what SignalProcessor class is used to handle Django's signals and keep the search index up-to-date.

An example:

```
ELASTICSEARCH_DSL_SIGNAL_PROCESSOR = 'django_elasticsearch_dsl.signals.
↪RealTimeSignalProcessor'
```

Defaults to `django_elasticsearch_dsl.signals.RealTimeSignalProcessor`.

Options: `django_elasticsearch_dsl.signals.RealTimeSignalProcessor` `django_elasticsearch_dsl.signals.CelerySignalProcessor`

In this `CelerySignalProcessor` implementation, Create and update operations will record the updated data primary key from the database and delay the time to find the association to ensure eventual consistency. Delete operations are processed to obtain associated data before database records are deleted. And celery needs to be pre-configured in the django project, for example *Using Celery with Django <https://docs.celeryq.dev/en/stable/django/first-steps-with-django.html>*.

You could, for instance, make a `CustomSignalProcessor` which would apply update jobs as your wish.

5.5 ELASTICSEARCH_DSL_PARALLEL

Default: `False`

Run indexing (populate and rebuild) in parallel using ES' parallel_bulk() method. Note that some databases (e.g. sqlite) do not play well with this option.

CHAPTER 6

Management Commands

Delete all indices in Elasticsearch or only the indices associate with a model (`--models`):

```
$ search_index --delete [-f] [--models [app[.model] app[.model] ...]]
```

Create the indices and their mapping in Elasticsearch:

```
$ search_index --create [--models [app[.model] app[.model] ...]]
```

Populate the Elasticsearch mappings with the Django models data (index need to be existing):

```
$ search_index --populate [--models [app[.model] app[.model] ...]] [--parallel] [--
↪refresh]
```

Recreate and repopulate the indices:

```
$ search_index --rebuild [-f] [--models [app[.model] app[.model] ...]] [--parallel] [-
↪-refresh]
```

Recreate and repopulate the indices using aliases:

```
$ search_index --rebuild --use-alias [--models [app[.model] app[.model] ...]] [--
↪parallel] [--refresh]
```

Recreate and repopulate the indices using aliases, but not deleting the indices that previously pointed to the aliases:

```
$ search_index --rebuild --use-alias --use-alias-keep-index [--models [app[.model]␣
↪app[.model] ...]] [--parallel] [--refresh]
```

CHAPTER 7

Contributing

We are glad to welcome any contributor.

Report bugs or propose enhancements through *github bug tracker*

github bug tracker: https://github.com/sabricot/django-elasticsearch-dsl/issues

If you want to contribute, the code is on github: https://github.com/sabricot/django-elasticsearch-dsl

7.1 Testing

You can run the tests by creating a Python virtual environment, installing the requirements from `requirements_test.txt` (`pip install -r requirements_test`):

```
$ python runtests.py
```

For integration testing with a running Elasticsearch server:

```
$ python runtests.py --elasticsearch [localhost:9200]
```

7.2 TODO

- Add support for –using (use another Elasticsearch cluster) in management commands.
- Add management commands for mapping level operations (like update_mapping. . . .).
- Generate ObjectField/NestField properties from a Document class.
- More examples.
- Better `ESTestCase` and documentation for testing

CHAPTER 8

Indices and tables

- genindex
- modindex
- search