djangoes Documentation

Release 0.3.0

Florian Strzelecki

November 20, 2015

Contents

1 Connections to ElasticSearch **1**
1.1 Threading and multiprocessing . . . 1

2 Perform queries **3**
2.1 Search . . . 3
2.2 Single index operation . . . 4
2.3 Advanced usage . . . 4
2.4 Multiple connections and indices . . . 4

3 Configure your django project **7**
3.1 Settings . . . 7
3.2 Timeout and retry on error . . . 8

4 Backends **11**
4.1 Available backends . . . 11
4.2 Custom backends . . . 12

5 Package documentation **13**
5.1 backends . . . 13
5.2 test . . . 16

6 Install **19**

7 Short introduction **21**

8 Indices and tables **23**

Python Module Index **25**

CHAPTER 1

Connections to ElasticSearch

The `djangoes` package provides a simple way to use and to access the default connection to your ElasticSearch server. In any of your python file of your Django project, simply import `connection` like this:

```
from djangoes import connection
```

Then in any function or method you need to perform a query, you can use the methods from the ElasticSearch python library by using this `connection` object:

```
def search_blog_entries(words):
    """Search for all blog entries with ``words`` found in entry body."""
    doc_type = 'entry'
    search = {
        'query': {
            'term': {
                'text': words
            }
        }
    }
    result = connection.search(doc_type, search)

    # Result from ES "as is", not modified by djangoes.
    return result.get('hits', {}).get('hits', [])
```

If you want to select a specific connection, you can import `connections` instead:

```
from djangoes import connections
```

And it can be used like this:

```
# in some function or method
conn = connections['connection_alias']
```

In fact, the `connection` object is a simple proxy to the default connection, simply named `default`:

```
>>> from djangoes import connection, connections
>>> connection == connections['default']
True
```

1.1 Threading and multiprocessing

Using `connection` or `connections` is **thread-safe**, but you should never use a connection object itself in multiple threads. If you need to "share" a connection from one thread to another, simply use its alias and get it using

`connections[alias]` into the threaded code.

The same way, you shoulw **never** share a connection object between multiple process (either with multiprocessing or forking), and instead use the `connection` shortcut or `connections[alias]` to get any connection.

Connection's methods are not thread or multi-process safe by themselves, and an unappropriate usage may end in unexpected behavior.

CHAPTER 2

Perform queries

Now that you have a connection to your ElasticSearch servers, you may want to perform queries. The connection object has an equivalent interface as the official `elasticsearch.client.Elasticsearch` class, but simplifies most method by removing the `index` parameter.

See also:

The official ElasticSearch API documentation.

2.1 Search

The most common query to perform is the `search` - which is expected for a software named "ElasticSearch". When using `djangoes`, performing a search query is as simple as using the official `elasticsearch` library. For example, to perform a search query on the configured index, using the document type "blog entry":

```
>>> from djangoes import connection
>>> search = {'query': {'match_all': {}}}
>>> result = connection.search(doc_type='blog_entry', body=search)
>>> result.get('hits', {}).get('hits', [])
[ ... list of all indexed blog entries ... ]
```

OK, but why the last line with the chained `get`? Better to read both documentations of the search method, and the search API - the last giving a sample response example:

```
{
    "_shards":{
        "total" : 5,
        "successful" : 5,
        "failed" : 0
    },
    "hits":{
        "total" : 1,
        "hits" : [
            {
                "_index" : "twitter",
                "_type" : "tweet",
                "_id" : "1",
                "_source" : {
                    "user" : "kimchy",
                    "postDate" : "2009-11-15T14:12:12",
                    "message" : "trying out Elasticsearch"
                }
```

```
            }
        ]
    }
}
```

As you can see, the results is a `dict` that contains meta information about the request (stats on shards), and the `hits`, a dict that contains the results of the query: the total number of documents that match the search query, and the list of documents for the current page.

2.2 Single index operation

Not all operation can be performed accross multiple indices: `get`, `create` or `update` queries are single index operations. They can not be performed on a list of indices, nor on an alias mapped to more than one index.

2.3 Advanced usage

To perform any advanced queries, such as getting the list of aliases for an index, the `client` attribute is available on each connection: it is the underlying client implementation, ie. an instance of `from elasticsearch.client.Elasticsearch`.

> **Warning:** At the moment, the `client` attribute is not well documented as the behavior of the backend is supposed to change in a near future, with a more stable API.

2.4 Multiple connections and indices

There are way too many possible configurations for your application, your ElasticSearch servers and indices. Therefore, `djangoes` tries to stay agnostic about your way of using ElasticSearch.

Let's see an example of configuration and how to use it.

See also:

Configure your django project

2.4.1 Search & Single-operation configuration

In this situation, one wants to perform search queries on multiple indices, *and* to insert documents into these indices. Let's start by the `ES_SERVERS` configuration:

```
ES_SERVERS = {
    'default': {
        'HOSTS': ['localhost:9200'],
        'INDICES': ['categories', 'brands']
    },
    'categories': {
        'HOSTS': ['localhost:9200'],
        'INDICES': ['categories']
    },
    'brands': {
        'HOSTS': ['localhost:9200'],
```

```
        'INDICES': ['brands']
    }
}
```

Now, we need to configure these indices:

```
ES_INDICES = {
    'categories': {
        'SETTINGS': {
            // Category-specific index settings
        }
    },
    'brands': {
        'SETTINGS': {
            // Brand-specific index settings
        }
    }
}
```

And then the magic happens:

```
>>> from djangoes import connection
>>> results = connection.search(query)
>>> results.get('hits', {}).get('hits', [])
[ some_category, some_brand, some_other_category, ... ]
```

This is possible because the `djangoes` client uses the `connection.indices` property, which is the list of aliases, or index names if no alias is configured (which is our case here):

```
>>> connection.indices
['categories', 'brands']
```

Now, we still need to insert documents. For this, we'll use the other connections:

```
>>> from djangoes import connections
>>> categories = connections['categories']
>>> brands = connections['brands']
>>> categories.create(doc_id, doc_category)
{ ... result of the create action ... }
>>> brands.create(doc_id, doc_brand)
{ ... result of the create action ... }
```

As you can see, each connection has a different value for its `indices` attribute:

```
>>> categories.indices
['categories']
>>> brands.indices
['brands']
```

Therefore, you can handle this specific case - and it's only one of the many possible solutions.

Note: It's not simple to handle this case, as there are many ways to do it. At the moment, `djangoes` does not provide a really simple solution. This may change in a near future.

CHAPTER 3

Configure your django project

After installing `djangoes`, you need to configure your django project settings with two variables:

- `ES_SERVERS`: configure connections to ElasticSearch servers,
- `ES_INDICES`: configure ElasticSearch indices used by connections.

The main idea behind this separation is to configure the connections, the way to access the ElasticSearch API, and the indices separatly, where the documents are stored and how to access to them, then to decide what indices each connection will use.

For example, you can have two connections, one for each host, and both use the same index configuration.

3.1 Settings

ES_SERVERS

The setting `ES_SERVERS` is a dict, where each key is a connection configuration alias (its name), and each value is a dict that describes one connection. By default, there is one connection named `default` - the same way there is a `default` database connection alias in Django.

The keys expected in a connection dict are:

- `HOSTS`: a hosts configuration, as expected by elasticsearch-py,
- `ENGINE`: a string giving the class path to the engine backend class,
- `INDICES`: a `list` of index alias as found in `ES_INDICES`,
- `PARAMS`: a `dict` used as keyword arguments to instanciate the backend class.

Example:

```
ES_SERVERS = {
    'default': {
        'HOSTS': ['es_host_1', 'es_host_2'],
        'ENGINE': 'djangoes.backends.elasticsearch.SimpleHttpClient',
        'INDICES': ['index_1']
    }
}
```

See also:

Backends

The *Backends* chapter gives more information about the available backends, how they work and how to build yours.

ES_INDICES

The setting `ES_INDICES` is a dict, where each key is an index configuration alias (its name as used by connections in `ES_SERVERS` in their `INDICES` option), and each value is a dict that describes one index. By default, no index are defined.

The expected keys are:

- `NAME`: a `string`, its index name, by default it will be its configuration alias if not explicitly given,
- `ALIASES`: a `list` of alias names, by default an empty `list`,
- `SETTINGS`: an optionnal `dict` used to describe the index's settings when creating this index.
- `TESTS`: a `dict` used to configure index when testing.

Example:

```
ES_INDICES = {
    'index_1': {
        'NAME': 'real_index_name',
        'ALIASES': ['index_catalog', 'index_public'],
    }
}
```

3.1.1 The `SETTINGS` parameter

Each index can have its own configuration: analyzers, tokenizers, and other index-specific settings. `djangoes` uses these settings in its test-case methods to create the test indices.

You might also use it in your own code thanks to the `get_indices_with_settings()` method:

```
>>> indices_with_settings = connection.get_server_indices()
>>> for index_name, settings_body in indices_with_settings.items():
...     connection.client.indices.create(index_name, settings_body)
```

3.2 Timeout and retry on error

Timeout configuration and management can be very important for your application, and it can become complicated to understand which parameters are available, and what they exactly mean - thus how to configure them.

As djangoes uses the official ElasticSearch python library to implement its client engines, it allows to configure the behavior on error caused by timeout: should the client retry on another server on timeout or not? How long a server should be marked as dead after a timeout? How many time should the client retry after an error?

In `ES_SERVERS`, each connection has a `PARAMS` key that contains the keyword arguments that will be given to the `ENGINE` backend class. Some of these arguments, described below, allow to control the behavior after a timeout or a connection error.

max_retries

Maximum number of retry after an error before a request raise an error.

It means that, when performing a request, the client will try as many time as `max_retries` before it raises an error.

It won't retry on client error, such as invalid request, but it will retry on another host if one is not reachable.

By default, it does not retry after a timeout error.

timeout
The time (in seconds) until a request to a server raises a timeout error.

retry_on_timeout
Indicates if the client must retry after a timeout or not. By default the client won't retry after a timeout, and will raise directly.

dead_timeout
Number of seconds a connection should be retired for after a failure, increases on consecutive failures

timeout_cutoff
Number of consecutive failures after which the timeout doesn't increase.

Example:

```
ES_SERVERS = {
    'default': {
        'HOSTS': ['host_1', 'host_2']
        'PARAMS': {
            'timeout': 1,
            'retry_on_timeout': True,
            'max_retries': 3
        }
    }
}
```

In this example, a request will raise a timeout error after 1 second, but the client will retry at most 3 times before raising a connection error itself.

See also:

ElasticSearch Transport documentation gives information about the behavior after an error (retry or not), and the ConnectionPool documentation gives information about timeout configuration.

CHAPTER 4

Backends

4.1 Available backends

Djangoes provides only simple and basic backends: they use transport and connection classes provided by default by `elastichsearch-py`. They aim to provide a set of methods easy to use, as they will automatically use the configured indices.

4.1.1 ElasticSearch backends

Each backend uses one of the connection class provided by `elastichsearch-py`:

- `elasticsearch.connection.http_urllib3.Urllib3HttpConnection`
- `elasticsearch.connection.http_requests.RequestsHttpConnection`
- `elasticsearch.connection.thrift.ThriftConnection`
- `elasticsearch.connection.memcached.MemcachedConnection`

To configure a backend, simply add the expected keyword arguments in the `PARAMS` key of the connection configuration dict.

See also:

All connection classes used by these backends are describe in the official documentation of `elasticsearch-py`.

`SimpleHttpBackend`

The backend `djangoes.backends.elasticsearch.SimpleHttpBackend` uses the connection class used by default by the `Transport` class: `elasticsearch.connection.http_urllib3.Urllib3HttpConnection`.

Each query will be performed with an HTTP request, using the `urllib3` library.

`SimpleRequestsHttpBackend`

The backend `djangoes.backends.elasticsearch.SimpleRequestsHttpBackend` uses the connection class `elasticsearch.connection.http_requests.RequestsHttpConnection`.

Each query will be performed with an HTTP request, using the `requests` library (also known as "HTTP for human").

SimpleThriftBackend

The backend `djangoes.backends.elasticsearch.SimpleThriftBackend` uses the connection class `elasticsearch.connection.thrift.ThriftConnection`.

Each query will be performed using the Thrift protocol.

SimpleMemcachedBackend

The backend `djangoes.backends.elasticsearch.SimpleMemcachedBackend` uses the connection class `elasticsearch.connection.memcached.MemcachedConnection`.

Each query will be performed using memcached.

4.2 Custom backends

Create a custom backend for your application is as easy as subclassing the abstract class, and implement your own methods.

> **Warning:** The backend interface is not yet stable. It should be, as soon as possible, but not yet. So don't rush in the code without a look at the current available backend classes.

A backend is expected to subclass the `djangoes.backends.abstracts.Base` class. Then, its `__init__` method is expected to accept these three parameters:

- `alias`: the connection's alias. It is the key used in `ES_SERVERS` to configure the connection using this backend.
- `server`: the configuration dict of the connection, as found in `ES_SERVERS`, where all undefined values are replaced by the defaults.
- `indices`: the list of configuration dict of the connection's indices, as found in `ES_INDICES`, where all undefined values are replaced by the defaults.

4.2.1 Extend ElasticSearch backends

The built-in `djangoes` backends are all based on an abstract class: `djangoes.backends.elasticsearch.BaseElasticsea`
This class conveniently subclass the abstract base class, and gives two entry point to override its behavior:

- `transport_class`: the transport class used to configure the `elasticsearch-py` client.
- `connection_class`: the connection class used by the transport class.

If you are already familiar with the transport class and the connection classes described in the `elasticsearch-py` library documentation, you should not have any issue with finding your way.

CHAPTER 5

Package documentation

Djangoes package aims to provide a simple way to integrate ElasticSearch.

This package mimics the behavior of the Django database configuration layer, using project settings and a global connections handler.

The simplest way to use `djangoes` is to import `djangoes.connection` and to perform queries with it:

```
>>> from djangoes import connections
>>> conn = connections['conn_name']
>>> result = conn.search(...)
```

The `ConnectionHandler.load_backend()` is called whenever a connection is requested in the application, which will then call multiple methods to ensure default values and test values.

There is a shortcut to get the default connection:

```
>>> from djangoes import connection, connections
>>> connection == connections['default']
True
>>> result = connection.search(...)
```

It works exactly like getting the `default` connection from `connections`.

Note: This module is based on the `django.db` module, which is quite simple in its way to deal with connections. The `djangoes` package hope to stay as simple as possible for everyone, and to take benefit from the hard works that make Django a great framework.

5.1 backends

Module djangoes.backends provides backend implementation.

5.1.1 backends.abstracts

Module djangoes.backends.abstracts provides abstract classes for backends.

All backends are expecting to subclass these abstract classes and to implement their behaviors.

class Base (*alias*, *server*, *indices*)

ElasticSearch backend wrapper base.

indices
List of names to use to perform ElasticSearch queries.

For each configured index, a connection will use either its index name, or its list of aliases if at least one is defined.

For example, if a connection uses 2 indices, one with only the `index` index, and the second one with the `index_2` index and an alias `alias`, the result `indices` will be `['index', 'alias']`.

index_names
List of names of all configured indices, without their aliases.

It is the same list of `indices` but where indices are not replaced by their aliases.

It is particulary useful when indices need to be created for example.

alias_names
List of names of all configured indices's aliases, without their indices.

It is the same list of `indices` but where only aliases are presents.

It is particulary useful when aliases need to be created for example.

configure_client()
Configure the ElasticSearch client.

get_alias_names()
Build and return the list of alias names.

This create a list of unique alias names, without using their indices. It can be useful to get alias names instead of their usage names, as given by `indices` as it would gives index names when no alias is configured - which is not always what is needed.

get_index_names()
Build and return the list of index names.

This create a list of unique index names, without using their aliases. It can be useful to get index names instead of their usage names, as given by `indices`, for example when one wants to create them.

get_indices()
Build the list of indices or aliases used to query ElasticSearch.

This creates a list composed of index names or alias names. If an index defined aliases, these aliases will be used instead of its own name.

get_indices_with_settings()
Build and return a dict of indices with their settings.

This create a dict where each key is a index name, and each value is the index key's settings (as used to created the index). It is useful when one wants to create an index with its settings for the given connection.

5.1.2 backends.elasticsearch

Connection backends based on the elasticsearch official python library.

This module aims to contain only basic backend classes using the ElasticSearch official python library and its basics transport classes with few changes.

Each backend implements the expected behavior defined in the base class - that is to say: they provide methods that don't need to get an index or an alias as argument to perform requests (when applicable).

class BaseElasticsearchBackend(*alias*, *server*, *indices*)

Base connection wrapper based on the ElasticSearch official library.

It uses two entry points to configure the underlying connection:

- `transport_class`: the transport class from `elasticsearch`. By default `elasticsearch.transport.Transport`.
- `connection_class`: the connection class used by the transport class. It's undefined by default, as it is on the subclasses to provide one.

If any of these elements is not defined, an `ImproperlyConfigured` error will be raised when the backend will try to configure the client.

configure_client()

Instantiate and configure the ElasticSearch client.

It simply takes the given HOSTS list and uses PARAMS as the keyword arguments of the ElasticSearch class.

The client's transport_class is given by the class attribute `transport_class`, and the connection class used by the transport class is given by the class attribute `connection_class`.

An `ImproperlyConfigured` exception is raised if any of these elements is undefined.

transport_class

alias of `Transport`

class SimpleHttpBackend(*alias*, *server*, *indices*)

Connection backend using the `urllib3` connection class.

connection_class

alias of `Urllib3HttpConnection`

class SimpleMemcachedBackend(*alias*, *server*, *indices*)

Connection backend using the Memcache connection class.

As describe in the `MemcachedConnection`, a plugin must be installed in the ElasticSearch cluster in order to work.

connection_class

alias of `MemcachedConnection`

class SimpleRequestsHttpBackend(*alias*, *server*, *indices*)

Connection backend using the HTTP for Human request connection class.

connection_class

alias of `RequestsHttpConnection`

class SimpleThriftBackend(*alias*, *server*, *indices*)

Connection backend using the Thrift experimental connection class.

connection_class

alias of `ThriftConnection`

class SimpleHttpBackend(*alias*, *server*, *indices*)

Connection backend using the `urllib3` connection class.

connection_class

alias of `Urllib3HttpConnection`

5.2 test

5.2.1 test.runner

Tests runners for Django projects with `djangoes`.

When using the built-in Django admin `test` command, the simplest way to integrate `djangoes` is to configure the `TEST_RUNNER` settings, using the runner provided by `djangoes`:

```
TEST_RUNNER = 'djangoes.test.runner.DiscoverRunner'
```

class DiscoverRunner(*pattern=None*, *top_level=None*, *verbosity=1*, *interactive=True*, *failfast=False*, ***kwargs*)

Unittest Runner with Django and ElasticSearch.

Setup ElasticSearch connections in order to run tests with the test settings and not the developement/production settings.

When using djangoes in a Django project, it requires to define the settings option `TEST_RUNNER` to `djangoes.test.runner.DiscoverRunner` to allow the tests with djangoes and ElasticSearch to work properly.

5.2.2 test.testcases

TestCase classes for ElasticSearch in Django.

These classes combine Django test case classes with djangoes mixin in order to replace them in a Django project with ElasticSearch.

Instead of doing:

```
from django.test.testcases import SimpleTestCase
```

One can do:

```
from djangoes.test.testcases import SimpleTestCase
```

It works the same way for `TransactionTestCase` and `TestCase`.

class SimpleTestCase(*methodName='runTest'*)

Simple test case with Django and ElasticSearch.

Automatically create the indices for all configured ElasticSearch connections, combined with the setup & tear down of the Django `SimpleTestCase` test case class.

class TestCase(*methodName='runTest'*)

Test case with Django and ElasticSearch.

Automatically create the indices for all configured ElasticSearch connections, combined with the setup & tear down of the Django `TestCase` test case class.

class TransactionTestCase(*methodName='runTest'*)

Transaction test case with Django and ElasticSearch.

Automatically create the indices for all configured ElasticSearch connections, combined with the setup & tear down of the Django `TransactionTestCase` test case class.

5.2.3 test.utils

Utility functions for testing purpose with `djangoes`.

setup_djangoes()

Setup ElasticSearch connections with `djangoes` for testing purpose.

When testing with ElasticSearch, used indices must not be the same as the one used for live settings, ie. tests must use the TEST settings.

This function takes care of replacing each used index name by its appropriate test name.

connections

Module-level attribute, instance of `ConnectionHandler`.

It can be considered as a singleton: it is the default connections handler to use with `djangoes`. It is instantiated at import with the default arguments.

Therefore, this object will automatically use the settings of your django project: `ES_SERVERS` and `ES_INDICES`.

class ConnectionHandler(*servers=None*, *indices=None*)

Handle connections to ElasticSearch.

Based on django.db.utils.ConnectionHandler, it aims to be an interface to integrate ElasticSearch connections in the same way database connections are integrated in Django.

However, instead of relaying on one setting variable, it needs two:

- *servers*: ElasticSearch clusters connections settings (host, port, etc.)
- *indices*: indices (or indexes) settings (name, aliases, analyzers, etc.)

These two will be defined in the django project settings with *ES_SERVERS* and *ES_INDICES*.

all()

Return all configured connection object.

It is a shortcut method to get all connections instead of manually doing a list-comprehension each time all connections are needed.

check_for_multiprocess()

Reset connections if PID has changed.

When using multi-processing (or fork), one may want to use a connection already used by the main process. Therefore, we need to make sure we are not sharing connections between multiple process.

This could happen because a fork on Linux won't copy an object after a fork until it is modified. The read-only mode will "share" connections and that's not what we want.

ensure_index_defaults(*alias*)

Put the defaults into the settings dictionary for *alias*.

ensure_server_defaults(*alias*)

Put the defaults into the settings dictionary for *alias*.

get_server_indices(*server*)

Prepare and return a given server's indices settings.

Do not validate if the given server is available in `self.servers`: it is expected to find an `INDICES` key into *server* and that's all.

It is expected to find indices configured with the same name in `self.indices`.

load_backend(*alias*)
Prepare and load a backend for the given alias.

prepare_index_test_settings(*alias*)
Make sure the test settings are available in *TEST*.

prepare_server_test_settings(*alias*)
Make sure the test settings are available in *TEST*.

Warning: This project is not production-ready yet, and it's published as Alpha version on Pypi. It works, it can be used, but its internal and public interfaces are not stable yet, and may change in a near future without warnings. As soon as the application is considered stable, this warning will be removed and the package published on Pypi will be marked as beta, then stable/production ready.
Deprecration warnings will be used, and a release cycle will be exposed, with all the version management we all wish to have.

CHAPTER 6

Install

To install `djangoes` and its dependencies, the simple way is to use pip:

```
$ pip install djangoes django elasticsearch
```

You should always use `pip` to install `djangoes`.

You will also need an ElasticSearch server. See the official documentation for that, but if you are running on Debian or Ubuntu, it will be as simple as adding a repository to your sources list and shoot an `apt-get install` in your favorite shell.

CHAPTER 7

Short introduction

When you have installed `djangoes`, you can configure your Django project with two news settings, like this:

```
# in your settings file
ES_SERVERS = {
    'default': {
        'HOSTS': ['localhost', ],
        'INDICES': ['my_index'],
    }
}

ES_INDICES = {
    'my_index': {
        'NAME': 'index_dev',
    }
}
```

See also:

Configure your django project

Then you can build your first view and use `djangoes.connection` to query ElasticSearch:

```
# in a views.py
from django.shortcuts import render
from djangoes import connection


def search_blog_entries(request):
    search_term = request.GET['q']
    query = {
        'query': {
            'match': {
                'text': search_term
            }
        }
    }
    result = connection.search(doc_type='entry', body=query)
    return render(request, 'search/results.html', {'results': result})
```

See also:

Connections to ElasticSearch and *Perform queries*

And finally in your template, you can display the result with this:

```
<h1>Blog post found</h1>

{% for hit in results.hits.hits %}
    <article>
        <h2>{{hit._source.title}}</h2>
        {{hit._source.text}}
    </article>
{% endfor %}
```

Note that this example uses the raw result, without any specific modification. It's because `djangoes` provides the connection layer only - everything else remains up to the developer to decide (for example by using the official DSL library, named elasticsearch-dsl).

CHAPTER 8

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index

b

`djangoes.backends`, 13
`djangoes.backends.abstracts`, 13
`djangoes.backends.elasticsearch`, 14

d

`djangoes`, 13

t

`djangoes.test`, 16
`djangoes.test.runner`, 16
`djangoes.test.testcases`, 16
`djangoes.test.utils`, 17

A

alias_names (Base attribute), 14
all() (ConnectionHandler method), 17

B

Base (class in djangoes.backends.abstracts), 13
BaseElasticsearchBackend (class in djangoes.backends.elasticsearch), 14

C

check_for_multiprocess() (ConnectionHandler method), 17
configure_client() (Base method), 14
configure_client() (BaseElasticsearchBackend method), 15
connection_class (SimpleHttpBackend attribute), 15
connection_class (SimpleMemcachedBackend attribute), 15
connection_class (SimpleRequestsHttpBackend attribute), 15
connection_class (SimpleThriftBackend attribute), 15
ConnectionHandler (class in djangoes), 17
connections (in module djangoes), 17

D

DiscoverRunner (class in djangoes.test.runner), 16
djangoes (module), 13
djangoes.backends (module), 13
djangoes.backends.abstracts (module), 13
djangoes.backends.elasticsearch (module), 14
djangoes.test (module), 16
djangoes.test.runner (module), 16
djangoes.test.testcases (module), 16
djangoes.test.utils (module), 17

E

ensure_index_defaults() (ConnectionHandler method), 17
ensure_server_defaults() (ConnectionHandler method), 17
ES_INDICES (built-in variable), 7
ES_SERVERS (built-in variable), 7

G

get_alias_names() (Base method), 14
get_index_names() (Base method), 14
get_indices() (Base method), 14
get_indices_with_settings() (Base method), 14
get_server_indices() (ConnectionHandler method), 17

I

index_names (Base attribute), 14
indices (Base attribute), 13

L

load_backend() (ConnectionHandler method), 17

P

prepare_index_test_settings() (ConnectionHandler method), 18
prepare_server_test_settings() (ConnectionHandler method), 18

S

setup_djangoes() (in module djangoes.test.utils), 17
SimpleHttpBackend (class in djangoes.backends.elasticsearch), 15
SimpleMemcachedBackend (class in djangoes.backends.elasticsearch), 15
SimpleRequestsHttpBackend (class in djangoes.backends.elasticsearch), 15
SimpleTestCase (class in djangoes.test.testcases), 16
SimpleThriftBackend (class in djangoes.backends.elasticsearch), 15

T

TestCase (class in djangoes.test.testcases), 16
TransactionTestCase (class in djangoes.test.testcases), 16
transport_class (BaseElasticsearchBackend attribute), 15