Seeker Documentation

Release 1.0

Dan Watson

May 30, 2017

Contents

1 Configuration **3**
- 1.1 Seeker Settings . . . 3
- 1.2 Model Indexing Middleware . . . 4

2 Mappings **5**
- 2.1 Our Example Model . . . 5
- 2.2 Basic Documents . . . 5
- 2.3 The Registry . . . 6
- 2.4 Customizing Field Mappings . . . 6
- 2.5 Indexing Data . . . 6
- 2.6 Customizing The Entire Data Mapping . . . 7
- 2.7 What Gets Indexed and How . . . 7
- 2.8 Non-Django Documents . . . 7
- 2.9 Module Documentation . . . 7

3 Views **9**
- 3.1 Basic View . . . 9
- 3.2 Customizing Facets . . . 9
- 3.3 Class Reference . . . 9

4 Utility Functions **11**
- 4.1 Module Reference . . . 11

5 Unit Tests **13**
- 5.1 Running Tests . . . 13

6 Indices and tables **15**

Contents:

CHAPTER 1

Configuration

Seeker Settings

SEEKER_INDEX

Default: `seeker`

The name of the ES index that should be used by default. This can be overridden per mapping.

SEEKER_DEFAULT_OPERATOR

Default: `AND`

The default operator to use when performing keyword queries. This can be overridden per view.

SEEKER_BATCH_SIZE

Default: `1000`

The default indexing batch size.

SEEKER_DEFAULT_FACET_TEMPLATE

Default: `seeker/facets/terms.html`

The default template to use when rendering facets. Can be overridden per facet.

SEEKER_MAPPING_MODULE

Default: `mappings`

The name of the python module to try to automatically import from each app. Setting to `False` or `None` will cause seeker to skip doing any automatic imports.

SEEKER_DEFAULT_ANALYZER

Default: `snowball`

The analyzer to use by default when creating `elasticsearch_dsl.String` fields. Also used by default in `SeekerView` to determine how query strings should be analyzed (it's important that queries are analyzed the same way as your data).

Model Indexing Middleware

For sites that want model instances to be automatically indexed when they are created, updated, or deleted, Seeker includes a `ModelIndexingMiddleware` that connects to Django's `post_save` and `post_delete` signals. To use it, simply add `seeker.middleware.ModelIndexingMiddleware` to your `MIDDLEWARE_CLASSES` setting above any middleware that might alter model instances you want indexed.

Models are not automatically indexed when outside of a request cycle (with `ModelIndexingMiddleware` installed), to prevent unwanted or premature indexing during load scripts, bulk updates, etc. Instances may be indexed manually using `seeker.index`. If automatic updating is desired outside of the request cycle, it is possible to simply instantiate `ModelIndexingMiddleware` and keep a reference to it. The class connects to `post_save` and `post_delete` when created, so you may do something like:

```
from seeker.middleware import ModelIndexingMiddleware
middleware = ModelIndexingMiddleware()
# Update your model instances as necessary, they will be automatically indexed.
del middleware
```

CHAPTER 2

Mappings

Our Example Model

For the purposes of this document, take the following models:

```
class Author (models.Model):
    name = models.CharField(max_length=100)
    age = models.IntegerField()

    def __unicode__(self):
        return self.name

class Post (models.Model):
    author = models.ForeignKey(Author, related_name='posts')
    slug = models.SlugField()
    title = models.CharField(max_length=100)
    body = models.TextField()
    date_posted = models.DateTimeField(default=timezone.now)
    published = models.BooleanField(default=True)
```

Basic Documents

Documents are analogous to Django models, but for Elasticsearch instead of a database. For the simplest cases, you can let seeker define the document for you, indexing any field it can:

```
import seeker
from .models import Post

PostDoc = seeker.document_from_model(Post)
seeker.register(PostDoc)
```

Most built-in Django field types are automatically indexed, including `ForeignKey` and `ManyToManyField` (using their unicode representations).

The Registry

In order for seeker to know about a document for indexing purposes, you need to register it.

Customizing Field Mappings

You can specify how seeker builds the mapping for your model class in several ways:

```
import elasticsearch_dsl as dsl

class PostDoc (seeker.ModelIndex):
    # Custom field definition for existing field
    author = dsl.Object(properties={
        'name': seeker.RawString,
        'age': dsl.Integer(),
    })
    # New field not defined by the model
    word_count = dsl.Long()

    class Meta:
        mapping = seeker.build_mapping(Post, fields=('title', 'body'), exclude=('slug
↪',))

    @classmethod
    def queryset(cls):
        return Post.objects.select_related('author')
```

Think of `Meta.mapping` as the "base" set of fields, which you can then customize by defining them directly on the document class. Any field defined on your document class will take precedence over those built in `Meta.mapping` with the same name, and any new fields will be added to the mapping.

Notice in the example above that `author` is overridden to use Elasticsearch object type, and `word_count` is an extra field not defined by the `Post` model.

Indexing Data

When Seeker goes to index this document, it will automatically pull data from any model field (or property) with a matching name. So in this example, `title`, `body`, and `author` will automatically be sent for indexing, but you will need to generate `word_count` yourself. To do this, you can implement a `prepare_word_count` class method:

```
class PostDoc (seeker.ModelIndex):
    # ...

    @classmethod
    def prepare_word_count(cls, obj):
        return len(obj.body.split())
```

Alternatively, you could declare a `word_count` property on the `Post` model.

Customizing The Entire Data Mapping

If, for some reason, you need to customize the entire data mapping process, you may override the `serialize` class method:

```
class PostDoc (seeker.ModelIndex):
    # ...

    @classmethod
    def serialize(cls, obj):
        # Let seeker grab the field values it knows about from the model.
        data = super(PostDoc, cls).serialize(obj)
        # Manipulate the data from the default implementation. Or not.
        return data
```

The default implementation of `serialize` calls `seeker.mapping.serialize_object()` and `get_id`.

What Gets Indexed and How

When re-indexing a mapping, the process is as follows:

1. `seeker.mapping.ModelIndex.documents()` is called, and expected to yield a single dictionary at a time to index.
2. `seeker.mapping.ModelIndex.queryset()` is called to get the queryset of Django objects to index.
3. The resulting queryset is sliced into groups of `batch_size` (ordered by PK), to avoid a single large query.
4. For each object, `seeker.mapping.ModelIndex.should_index()` is called to determine if the object should be indexed. By default, all objects are indexed.
5. `seeker.mapping.ModelIndex.get_id()` and `seeker.mapping.ModelIndex.serialize()` are called to generate the ID and data sent to Elasticsearch for each object.

Non-Django Documents

It's possible to use seeker to build documents not associated with Django models. To do so, simply subclass `seeker.Indexable` instead of `seeker.ModelIndex`, and override `seeker.ModelIndex.documents`, like so:

```
class OtherDoc (seeker.Indexable):

    @classmethod
    def documents(cls, **kwargs):
        return [
            {'name': 'Dan Watson', 'comment': 'Hello wife.'},
            {'name': 'Alexa Watson', 'comment': 'Hello husband.'},
        ]
```

Module Documentation

CHAPTER 3

Views

Basic View

Seeker provides a Django class-based `SeekerView` that can be subclassed and customized for basic keyword searching and faceting. To get started, you might define a view hooked up to *PostMapping*:

```
from .mappings import PostDoc
import seeker

class PostSeekerView (seeker.SeekerView):
    document = PostDoc

urlpatterns = patterns('',
    url(r'^posts/$', PostSeekerView.as_view(), name='posts'),
)
```

By default, `SeekerView` renders a template named `seeker/seeker.html`, which can be customized through subclassing. The included template renders a fully-functional search page using Bootstrap and jQuery (hosted off CDNs).

Customizing Facets

TODO

Class Reference

CHAPTER 4

Utility Functions

Module Reference

CHAPTER 5

Unit Tests

Running Tests

The Seeker unit tests are based on Django's testing framework. The easiest way to get the tests running is to create a virtualenv, then:

```
cd tests
pip install -r requirements.txt
python manage.py test
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search