BetterWorks, Inc.

Summer 2015

Technical Blog Post: Elasticsearch

Author:
Alex Francis

August 7, 2015

Contents

1 Introduction

Bungiesearch is an open source repository created by Sparrho, a search and recommendation engine for scientific information based in the UK. Sparrho, in tandem with Elastic's newest module elasticsearch-dsl-py, combines the power of Elasticsearch with the expressiveness of Django and Python. In this blog post, I will discuss both open source projects in some technical detail, and address the challenges associated with utilizing these technologies in test and production environments here at BetterWorks.

2 Elasticsearch DSL

Elasticsearch DSL (`https://github.com/elastic/elasticsearch-dsl-py`) is a python library written by the creators of Elasticsearch that combines the flexibility of Elasticsearch with the expressiveness of Python. It is, quite simply, Pythonic "syntactic sugar" on top of Elasticsearch JSON DSL. Elasticsearch queries are dynamic and well-structured, but usually not suitable for widespread use in Python projects. Arbitrary constructions of JSON based on different filters and queries quickly become unmaintainable. Say, for example, you had a fairly simply Article model that had attribute 'description', and you wanted to construct a fairly simple fuzzy search query to match your scenario. The elasticsearch query JSON is below:

```
{
  'query': {
    'bool': {
      'should': [
        {
          'fuzzy_like_this_field': {
            'description': {
              'like_text': your_query,
              'fuzziness': 2
            }
          }
        }
      ]
    }
  }
}
```

This is obviously rigid and prone to developer error. It's non-extendible to other similar queries we may want to execute. It would fill documents with query structures that have little place in most environments.

Elasticsearch DSL provides the flexibility of Elasticsearch in a way that's Pythonic, and is well-maintained by a trusted group of Elastic developers. So what's the catch? The issue comes in the translation of Elasticsearch DSL to Django Python. Here at BetterWorks, our RESTful API uses Django's ORM, so we needed a connector. Elasticsearch DSL does provide a way to create a model-like wrapper around your documents, using the `DocType` class:

```
class Article(DocType):
        title = String(analyzer='snowball', fields={'raw': String(index='not_analyzed')})
        body = String(analyzer='snowball')
        tags = String(index='not_analyzed')
        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
```

But converting all of our Django business logic into a duplicate model exclusively for use in ES DSL seemed unreasonable. We needed something to bring together the worlds of Django and Elasticsearch, with the same Pythonic functionality in the new world of Elasticsearch DSL.

3 Bungiesearch

Bungiesearch seemed like the ideal fit for the job from the outset. The creators state, "Bungiesearch is a Django wrapper for elasticsearch-dsl-py. It inherits from elasticsearch-dsl-py's Search class, so all the fabulous features developed by the elasticsearch-dsl-py team are also available in Bungiesearch."

Bungiesearch has syntax that resembles Django's in many ways. Bungiesearch is filled with features and functionality that fill a manual, so I will try to concisely detail some of the magic it has produced without getting into most specifics.

The truly powerful thing about Bungiesearch is the code reusability and simplicity. These qualities are primarily derived from the concepts of search managers and search aliases, which I will discuss in more detail in the next couple of sections.

3.1 Bungiesearch Managers

Instead of managing objects with a Django Manager, Bungiesearch has it's own concept of managers, which extend the Django version itself through inheritance. This attaches nicely with models in any project. We chose to pursue separate "objects" and "search" managers, using a base Django manager and Bungiesearch Manager, respectively. A simple example follows:

```
class User(models.Model):
    name = models.TextField(db_index=True)
    user_id = models.TextField(blank=True, primary_key=True)
    description = models.TextField(blank=True)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(null=True)

    objects = Manager()
    search = BungiesearchManager()

    class Meta:
        app_label = 'core'
```

With a simple manager we can begin to construct, piecewise, powerful and dynamic searches that take into account arbitrary numbers of filters, queries, and custom parameters. Following the example from above, we can gain access to the User search manager by simple writing:

```
User.search
```

We can then gain access to a corresponding Bungiesearch object by attaching a search to that (yes, the nomenclature gets a bit conflicted here):

```
User.search.search
```

Remember, since Bungiesearch objects are just subclasses of Elasticsearch DSL's search object, we just gained access to all the power of Elasticsearch DSL via our manager! Attaching queries and filters elegantly just became easy. Let's say I want to fuzzy query the description, like I did in the clunky JSON query at the beginning of this post. Now, I can write:

```
User.search.search.query('fuzzy', description=your_query)
```

Cool, huh? This barely scratches the surface. Elasticsearch DSL supports a variety of features that allow you to manipulate your results as you need, from custom text analyzers and tokenizers to custom score functions on relevant results

in the index, which you can check out in the documentation, if interested.

These examples do raise a consequential question: "Sure this is simple, and you have constant access to powerful search managers, but how is this reusable? Wouldn't you need to construct queries from scratch each time, or at least perform lots of case checking?"
The answer is no, and the reason behind it segues perfectly into discussion of another Bungiesearch construct: search aliases.

3.2 Search Aliases

Search aliases are reusable search shortcuts that take an arbitrary number of parameters.

Let's pull an example from BetterWorks to bring this together — the main search functionality in the top search bar in our application. This handles a variety of query cases, and performs a series of operations in a small amount of space. If the query only has single character terms, we use a `match_phrase_prefix` elasticsearch query type. If this isn't the case, we construct a fuzzy query type instead. For the query itself, we also create a version of the query in which the stopwords (single characters terms) are split out. Finally, we boost the goals of a user's manager after the query has been formed using a custom score function, and then we paginate the search results sorted by relevance score.

The beauty of this is that search aliases attach to Bungiesearch objects and managers much like filters and queries, and the syntax resembles something like the below statement, which would return a list of the goal objects matching the query executed using the search alias described above:

```
Goal.search.search.bsearch_auto_query(your_query).execute()
```

The overall architecture is structured in the following way:

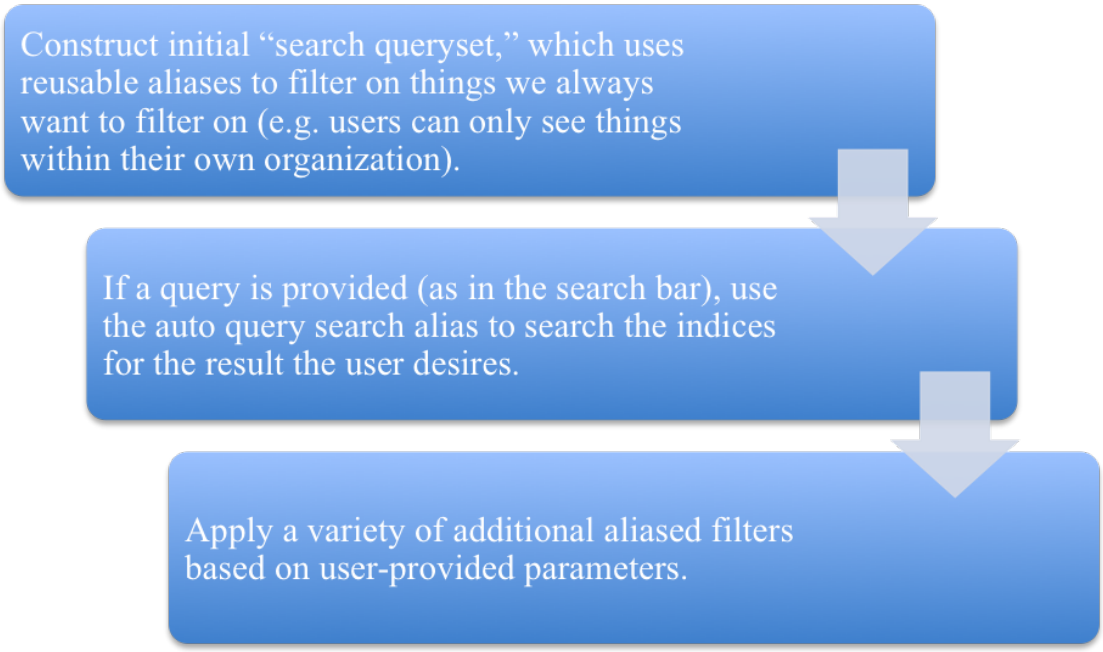

Figure 1: Django + Bungiesearch Search Architecture

As you can see, we build aliases into the entire process, which allows for simple, maintainable, Django-friendly code.

4 Other Technical Challenges and Takeaways

During the process of integrating Bungiesearch into the backend, we experienced a few major issues associated with maintaining high-performance code that matched previous functionality. I will detail one that was particularly important.

4.1 Asynchronous Signal Processing

Signal processing was a major technical challenge associated with Bungiesearch integration. Signal processors are responsible for synchronizing the (authoritative) database & the (non-authoritative) search index. This is critically important to any application that would use Bungiesearch: introducing delay between item creation and index updates severely harms search functionality. Luckily, Bungiesearch accounted for this from the outset, by connecting Django's signals (specifically

`models.db.signals.post_save` & `models.db.signals.post_delete`) to index update and delete functions (which interface directly with `elasticsearch-py`).

The only catches were that those functions were not really mutable in any way for the user, and didn't operate asynchronously. At BetterWorks, we have created our own distributed task queue using celery, and we needed to be able to wrap the signals from Bungiesearch in these tasks. It was this technical challenge that prompted a series of changes to the Bungiesearch project itself and led to the creation of `celery-bungiesearch` (`https://github.com/afrancis13/celery-bungiesearch`), which is designed to wrap custom celery tasks around custom signal processors. Implementing this successfully allowed us to continue to rely on our asynchronous task queue while integrating Bungiesearch into our architecture.