Last updated August 31st, 2021

# Django REST Framework and Elasticsearch

NT Nik Tomazic



In this tutorial, we'll look at how to integrate Django REST Framework (DRF) with Elasticsearch. We'll use Django to model our data and DRF to serialize and serve it. Finally, we'll index the data with Elasticsearch and make it searchable.

## What is Elasticsearch?

Elasticsearch is a distributed, free and open search and analytics engine for all types of data, including textual, numerical, geospatial, structured, and unstructured. It's known for its simple RESTful APIs, distributed nature, speed, and scalability. Elasticsearch is the central component of the Elastic Stack (also known as the ELK Stack), a set of free and open tools for data ingestion, enrichment, storage, analysis, and visualization.

Its use cases include:

1. Site search and application search
2. Monitoring and visualizing your system metrics
3. Security and business analytics
4. Logging and log analysis

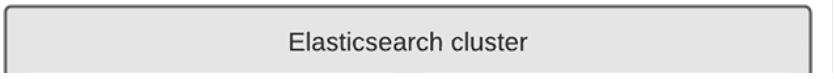
> To learn more about Elasticsearch check out What is Elasticsearch? from the official documentation.

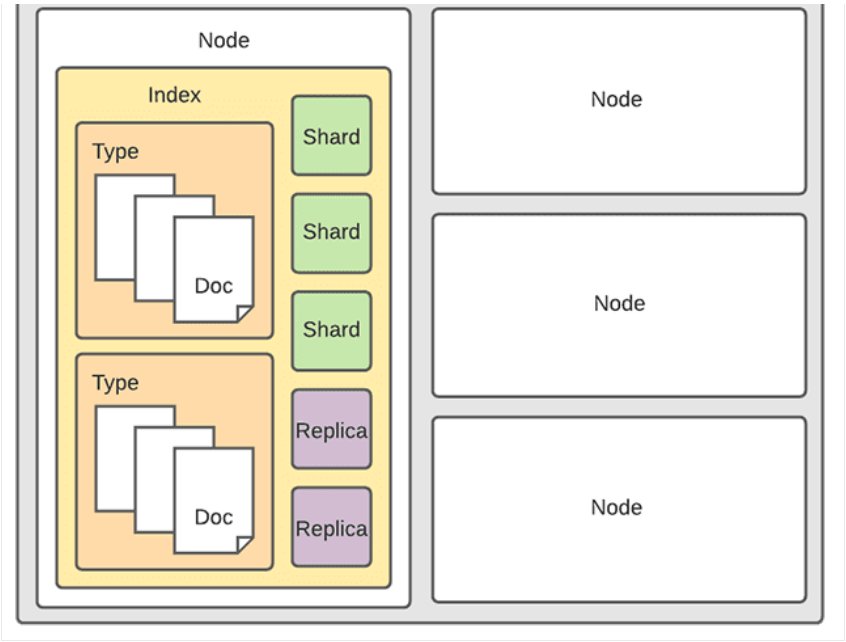
## Elasticsearch Structure and Concepts

Before working with Elasticsearch, we should get familiar with the basic Elasticsearch concepts. These are listed from biggest to smallest:

1. **Cluster** is a collection of one or more nodes.
2. **Node** is a single server instance that runs Elasticsearch. While communicating with the cluster, it:
    1. Stores and indexes your data
    2. Provides search
3. **Index** is used to store the documents in dedicated data structures corresponding to the data type of fields (akin to a SQL database). Each index has one or more shards and replicas.
4. **Type** is a collection of documents, which have something in common (akin to a SQL table).
5. **Shard** is an Apache Lucene index. It's used to split indices and keep large amounts of data manageable.
6. **Replica** is a fail-safe mechanism and basically a copy of your index's shard.
7. **Document** is a basic unit of information that can be indexed (akin to a SQL row). It's expressed in JSON, which is a ubiquitous internet data interchange format.
8. **Field** is the smallest individual unit of data in Elasticsearch (akin to a SQL column).

The Elasticsearch cluster has the following structure:






Curious how relational database concepts relate to Elasticsearch concepts?

| Relational Database | Elasticsearch |
| --- | --- |
| Cluster | Cluster |
| RDBMS Instance | Node |
| Table | Index |
| Row | Document |
| Column | Field |

> Review Mapping concepts across SQL and Elasticsearch for more on how concepts in SQL and Elasticsearch relate to one another.

# Elasticsearch vs PostgreSQL Full-text Search

With regards to full-text search, Elasticsearch and PostgreSQL both have their advantages and disadvantages. When choosing between them you should consider speed, query complexity, and budget.

PostgreSQL advantages:

1. Django support
2. Faster and easier to setup
3. Doesn't require maintenance

Elasticsearch advantages:

1. Optimized just for searching
2. Elasicsearch is faster (especially as the number of records increases)
3. Supports different query types (Leaf, Compound, Fuzzy, Regexp, to name a few)

If you're working on a simple project where speed isn't important you should opt for PostgreSQL. If performance is important and you want to write complex lookups opt for Elasticsearch.

Feedback

> For more on full-text search with Django and Postgres, check out the Basic and Full-text Search with Django and Postgres article.

# Project Setup

We'll be building a simple blog application. Our project will consist of multiple models, which will be serialized and served via Django REST Framework. After integrating Elasticsearch, we'll create an endpoint that will allow us to look up different authors, categories, and articles.

To keep our code clean and modular, we'll split our project into the following two apps:

1. `blog` - for our Django models, serializers, and ViewSets
2. `search` - for Elasticsearch documents, indexes, and queries

Start by creating a new directory and setting up a new Django project:

```
$ mkdir django-drf-elasticsearch && cd django-drf-elasticsearch
$ python3.9 -m venv env
$ source env/bin/activate

(env)$ pip install django==3.2.6
(env)$ django-admin.py startproject core .
```

After that, create a new app called `blog`:

```
(env)$ python manage.py startapp blog
```

Register the app in *core/settings.py* under `INSTALLED_APPS`:

```
# core/settings.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog.apps.BlogConfig', # new
]
```

# Database Models

Next, create `Category` and `Article` models in *blog/models.py*:

Feedback

```
# blog/models.py

from django.contrib.auth.models import User
from django.db import models


class Category(models.Model):
    name = models.CharField(max_length=32)
    description = models.TextField(null=True, blank=True)

    class Meta:
        verbose_name_plural = 'categories'

    def __str__(self):
        return f'{self.name}'


ARTICLE_TYPES = [
    ('UN', 'Unspecified'),
    ('TU', 'Tutorial'),
    ('RS', 'Research'),
    ('RW', 'Review'),
]


class Article(models.Model):
    title = models.CharField(max_length=256)
    author = models.ForeignKey(to=User, on_delete=models.CASCADE)
    type = models.CharField(max_length=2, choices=ARTICLE_TYPES, default='UN')
    categories = models.ManyToManyField(to=Category, blank=True, related_name='categories')
    content = models.TextField()
    created_datetime = models.DateTimeField(auto_now_add=True)
    updated_datetime = models.DateTimeField(auto_now=True)

    def __str__(self):
        return f'{self.author}: {self.title} ({self.created_datetime.date()})'
```

Notes:

1. `Category` represents an article category -- i.e, programming, Linux, testing.
2. `Article` represents an individual article. Each article can have multiple categories. Articles have a specific type -- `Tutorial`, `Research`, `Review`, or `Unspecified`.
3. Authors are represented by the default Django user model.

## Run Migrations

Make migrations and then apply them:

```
(env)$ python manage.py makemigrations
(env)$ python manage.py migrate
```

Register the models in *blog/admin.py*:

```
# blog/admin.py

from django.contrib import admin

from blog.models import Category, Article


admin.site.register(Category)
admin.site.register(Article)
```

## Populate the Database

Before moving to the next step, we need some data to work with. I've created a simple command we can use to populate the database.

Feedback

Create a new folder in "blog" called "management", and then inside that folder create another folder called "commands". Inside of the "commands" folder, create a new file called *populate_db.py*.

```
management
└── commands
    └── populate_db.py
```

Copy the file contents from populate_db.py and paste it inside your *populate_db.py*.

Run the following command to populate the DB:

```
(env)$ python manage.py populate_db
```

If everything went well you should see a `Successfully populated the database.` message in the console and there should be a few articles in your database.

# Django REST Framework

Now let's install `djangorestframework` using pip:

```
(env)$ pip install djangorestframework==3.12.4
```

Register it in our *settings.py* like so:

```
# core/settings.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog.apps.BlogConfig',
    'rest_framework', # new
]
```

Add the following settings:

```
# core/settings.py

REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.LimitOffsetPagination',
    'PAGE_SIZE': 25
}
```

We'll need these settings to implement pagination.

## Create Serializers

To serialize our Django models, we need to create a serializer for each of them. The easiest way to create serializers that depend on Django models is by using the `ModelSerializer` class.

*blog/serializers.py*:

Feedback

```
# blog/serializers.py

from django.contrib.auth.models import User
from rest_framework import serializers

from blog.models import Article, Category


class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ('id', 'username', 'first_name', 'last_name')


class CategorySerializer(serializers.ModelSerializer):
    class Meta:
        model = Category
        fields = '__all__'


class ArticleSerializer(serializers.ModelSerializer):
    author = UserSerializer()
    categories = CategorySerializer(many=True)

    class Meta:
        model = Article
        fields = '__all__'
```

Notes:

1. `UserSerializer` and `CategorySerializer` are fairly simple: We just provided the fields we want serialized.
2. In the `ArticleSerializer`, we needed to take care of the relationships to make sure they also get serialized. This is why we provided `UserSerializer` and `CategorySerializer`.

> Want to learn more about DRF serializers? Check out Effectively Using Django REST Framework Serializers.

## Create ViewSets

Let's create a ViewSet for each of our models in *blog/views.py*:

```
# blog/views.py

from django.contrib.auth.models import User
from rest_framework import viewsets

from blog.models import Category, Article
from blog.serializers import CategorySerializer, ArticleSerializer, UserSerializer


class UserViewSet(viewsets.ModelViewSet):
    serializer_class = UserSerializer
    queryset = User.objects.all()


class CategoryViewSet(viewsets.ModelViewSet):
    serializer_class = CategorySerializer
    queryset = Category.objects.all()


class ArticleViewSet(viewsets.ModelViewSet):
    serializer_class = ArticleSerializer
    queryset = Article.objects.all()
```

In this block of code, we created the ViewSets by providing the `serializer_class` and `queryset` for each ViewSet.

## Define URLs

Create the app-level URLs for the ViewSets:

Feedback

```
# blog/urls.py

from django.urls import path, include
from rest_framework import routers

from blog.views import UserViewSet, CategoryViewSet, ArticleViewSet

router = routers.DefaultRouter()
router.register(r'user', UserViewSet)
router.register(r'category', CategoryViewSet)
router.register(r'article', ArticleViewSet)

urlpatterns = [
    path('', include(router.urls)),
]
```

Then, wire up the app URLs to the project URLs:

```
# core/urls.py

from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('blog/', include('blog.urls')),
    path('admin/', admin.site.urls),
]
```

Our app now has the following URLs:

1. `/blog/user/` lists all users
2. `/blog/user/<USER_ID>/` fetches a specific user
3. `/blog/category/` lists all categories
4. `/blog/category/<CATEGORY_ID>/` fetches a specific category
5. `/blog/article/` lists all articles
6. `/blog/article/<ARTICLE_ID>/` fetches a specific article

## Testing

Now that we've registered the URLs, we can test the endpoints to see if everything works correctly.

Run the development server:

```
(env)$ python manage.py runserver
```

Then, in your browser of choice, navigate to http://127.0.0.1:8000/blog/article/. The response should look something like this:

Feedback

```
{
    "count": 4,
    "next": null,
    "previous": null,
    "results": [
        {
            "id": 1,
            "author": {
                "id": 3,
                "username": "jess_",
                "first_name": "Jess",
                "last_name": "Brown"
            },
            "categories": [
                {
                    "id": 2,
                    "name": "SEO optimization",
                    "description": null
                }
            ],
            "title": "How to improve your Google rating?",
            "type": "TU",
            "content": "Firstly, add the correct SEO tags...",
            "created_datetime": "2021-08-12T17:34:31.271610Z",
            "updated_datetime": "2021-08-12T17:34:31.322165Z"
        },
        {
            "id": 2,
            "author": {
                "id": 4,
                "username": "johnny",
                "first_name": "Johnny",
                "last_name": "Davis"
            },
            "categories": [
                {
                    "id": 4,
                    "name": "Programming",
                    "description": null
                }
            ],
            "title": "Installing latest version of Ubuntu",
            "type": "TU",
            "content": "In this tutorial, we'll take a look at how to setup the latest version of Ubuntu. Ubuntu
(/ʊˈbʊntuː/ is a Linux distribution based on Debian and composed mostly of free and open-source software. Ubuntu is
officially released in three editions: Desktop, Server, and Core for Internet of things devices and robots.",
            "created_datetime": "2021-08-12T17:34:31.540628Z",
            "updated_datetime": "2021-08-12T17:34:31.592555Z"
        },
        ...
    ]
}
```


Manually test the other endpoints as well.

# Elasticsearch Setup

Start by installing and running Elasticsearch in the background.

> Need help getting Elasticsearch up and running? Check out the Installing Elasticsearch guide. If you're familiar with Docker, you can simply run the following command to pull the official image and spin up a container with Elasticsearch running:
>
> ```
> $ docker run -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node" docker.elastic.co/elasticsearch
> /elasticsearch:7.14.0
> ```

To integrate Elasticsearch with Django, we need to install the following packages:

Feedback

1. elasticsearch - official low-level Python client for Elasticsearch
2. elasticsearch-dsl-py - high-level library for writing and running queries against Elasticsearch
3. django-elasticsearch-dsl - wrapper around elasticsearch-dsl-py that allows indexing Django models in Elasticsearch

Install:

```
(env)$ pip install elasticsearch==7.14.0
(env)$ pip install elasticsearch-dsl==7.4.0
(env)$ pip install django-elasticsearch-dsl==7.2.0
```

Start a new app called `search`, which will hold our Elasticsearch documents, indexes, and queries:

```
(env)$ python manage.py startapp search
```

Register the `search` and `django_elasticsearch_dsl` in *core/settings.py* under `INSTALLED_APPS`:

```
# core/settings.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django_elasticsearch_dsl', # new
    'blog.apps.BlogConfig',
    'search.apps.SearchConfig', # new
    'rest_framework',
]
```

Now we need to let Django know where Elasticsearch is running. We do that by adding the following to our *core/settings.py* file:

```
# core/settings.py

# Elasticsearch
# https://django-elasticsearch-dsl.readthedocs.io/en/latest/settings.html

ELASTICSEARCH_DSL = {
    'default': {
        'hosts': 'localhost:9200'
    },
}
```

> If your Elasticsearch is running on a different port, make sure to change the above settings accordingly.

We can test if Django can connect to the Elasticsearch by starting our server:

```
(env)$ python manage.py runserver
```

If your Django server fails, Elasticsearch is probably not working correctly.

# Creating Documents

Before creating the documents, we need to make sure all the data is going to get saved in the proper format. We're using `CharField(max_length=2)` for our article `type`, which by itself doesn't make much sense. This is why we'll transform it to human-readable text.

We'll achieve this by adding a `type_to_string()` method inside our model like so:

Feedback

```
# blog/models.py

class Article(models.Model):
    title = models.CharField(max_length=256)
    author = models.ForeignKey(to=User, on_delete=models.CASCADE)
    type = models.CharField(max_length=2, choices=ARTICLE_TYPES, default='UN')
    categories = models.ManyToManyField(to=Category, blank=True, related_name='categories')
    content = models.TextField()
    created_datetime = models.DateTimeField(auto_now_add=True)
    updated_datetime = models.DateTimeField(auto_now=True)

    # new
    def type_to_string(self):
        if self.type == 'UN':
            return 'Unspecified'
        elif self.type == 'TU':
            return 'Tutorial'
        elif self.type == 'RS':
            return 'Research'
        elif self.type == 'RW':
            return 'Review'

    def __str__(self):
        return f'{self.author}: {self.title} ({self.created_datetime.date()})'
```

Without `type_to_string()` our model would be serialized like this:

```
{
    "title": "This is my article.",
    "type": "TU",
    ...
}
```

After implementing `type_to_string()` our model is serialized like this:

```
{
    "title": "This is my article.",
    "type": "Tutorial",
    ...
}
```

Now let's create the documents. Each document needs to have an `Index` and `Django` class. In the `Index` class, we need to provide the index name and Elasticsearch index settings. In the `Django` class, we tell the document which Django model to associate it to and provide the fields we want to be indexed.

*blog/documents.py*:

Feedback

```
# blog/documents.py

from django.contrib.auth.models import User
from django_elasticsearch_dsl import Document, fields
from django_elasticsearch_dsl.registries import registry

from blog.models import Category, Article


@registry.register_document
class UserDocument(Document):
    class Index:
        name = 'users'
        settings = {
            'number_of_shards': 1,
            'number_of_replicas': 0,
        }

    class Django:
        model = User
        fields = [
            'id',
            'first_name',
            'last_name',
            'username',
        ]


@registry.register_document
class CategoryDocument(Document):
    id = fields.IntegerField()

    class Index:
        name = 'categories'
        settings = {
            'number_of_shards': 1,
            'number_of_replicas': 0,
        }

    class Django:
        model = Category
        fields = [
            'name',
            'description',
        ]


@registry.register_document
class ArticleDocument(Document):
    author = fields.ObjectField(properties={
        'id': fields.IntegerField(),
        'first_name': fields.TextField(),
        'last_name': fields.TextField(),
        'username': fields.TextField(),
    })
    categories = fields.ObjectField(properties={
        'id': fields.IntegerField(),
        'name': fields.TextField(),
        'description': fields.TextField(),
    })
    type = fields.TextField(attr='type_to_string')

    class Index:
        name = 'articles'
        settings = {
            'number_of_shards': 1,
            'number_of_replicas': 0,
        }

    class Django:
        model = Article
        fields = [
            'title',
            'content',
            'created_datetime',
            'updated_datetime',
        ]
```

Feedback

Notes:

1. In order to transform the article type, we added the `type` attribute to the `ArticleDocument`.
2. Because our `Article` model is in a many-to-many (M:N) relationship with `Category` and a many-to-one (N:1) relationship with `User` we needed to take care of the relationships. We did that by adding `ObjectField` attributes.

### Populate Elasticsearch

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

```
(env)$ python manage.py search_index --rebuild

Deleting index 'users'
Deleting index 'categories'
Deleting index 'articles'
Creating index 'users'
Creating index 'categories'
Creating index 'articles'
Indexing 3 'User' objects
Indexing 4 'Article' objects
Indexing 4 'Category' objects
```

> You need to run this command every time you change your index settings.

django-elasticsearch-dsl created the appropriate database signals so that your Elasticsearch storage gets updated every time an instance of a model is created, deleted, or edited.

## Elasticsearch Queries

Before creating the appropriate views, let's look at how Elasticsearch queries work.

We first have to obtain the `Search` instance. We do that by calling `search()` on our Document like so:

```
from blog.documents import ArticleDocument

search = ArticleDocument.search()
```

> Feel free to run these queries within the Django shell.

Once we have the `Search` instance we can pass queries to the `query()` method and fetch the response:

```
from elasticsearch_dsl import Q
from blog.documents import ArticleDocument


# Looks up all the articles that contain `How to` in the title.
query = 'How to'
q = Q(
     'multi_match',
     query=query,
     fields=[
         'title'
     ])
search = ArticleDocument.search().query(q)
response = search.execute()

# print all the hits
for hit in search:
    print(hit.title)
```

We can also combine multiple Q statements like so:

Feedback

```
from elasticsearch_dsl import Q
from blog.documents import ArticleDocument

"""
Looks up all the articles that:
1) Contain 'language' in the 'title'
2) Don't contain 'ruby' or 'javascript' in the 'title'
3) And contain the query either in the 'title' or 'description'
"""
query = 'programming'
q = Q(
     'bool',
     must=[
         Q('match', title='language'),
     ],
     must_not=[
         Q('match', title='ruby'),
         Q('match', title='javascript'),
     ],
     should=[
         Q('match', title=query),
         Q('match', description=query),
     ],
     minimum_should_match=1)
search = ArticleDocument.search().query(q)
response = search.execute()

# print all the hits
for hit in search:
    print(hit.title)
```

Another important thing when working with Elasticsearch queries is fuzziness. Fuzzy queries are queries that allow us to handle typos. They use the Levenshtein Distance Algorithm which calculates the distance between the result in our database and the query.

Let's look at an example.

By running the following query we won't get any results, because the user misspelled 'django'.

```
from elasticsearch_dsl import Q
from blog.documents import ArticleDocument

query = 'djengo'  # notice the typo
q = Q(
     'multi_match',
     query=query,
     fields=[
         'title'
     ])
search = ArticleDocument.search().query(q)
response = search.execute()

# print all the hits
for hit in search:
    print(hit.title)
```

If we enable fuzziness like so:

Feedback

```
from elasticsearch_dsl import Q
from blog.documents import ArticleDocument

query = 'djengo'  # notice the typo
q = Q(
     'multi_match',
     query=query,
     fields=[
         'title'
     ],
     fuzziness='auto')
search = ArticleDocument.search().query(q)
response = search.execute()

# print all the hits
for hit in search:
    print(hit.title)
```

The user will get the correct result.

> The difference between a full-text search and exact match is that full-text search runs an analyzer on the text before it gets indexed to Elasticsearch. The text gets broken down into different tokens, which are transformed to their root form (e.g., reading -> read). These tokens then get saved into the Inverted Index. Because of that, full-text search yields more results, but takes longer to process.

Elasticsearch has a number of additional features. To get familiar with the API, try implementing:

1. Your own analyzer.
2. Completion suggester - when a user queries 'j' your app should suggest 'johhny' or 'jess_'.
3. Highlighting - when user makes a typo, highlight it (e.g., Linuks -> *Linux*).

You can see all the Elasticsearch Search APIs here.

# Search Views

With that, let's create sime views. To make our code more DRY we can use the following abstract class in *search/views.py*:

Feedback

```
# search/views.py

import abc

from django.http import HttpResponse
from elasticsearch_dsl import Q
from rest_framework.pagination import LimitOffsetPagination
from rest_framework.views import APIView


class PaginatedElasticSearchAPIView(APIView, LimitOffsetPagination):
    serializer_class = None
    document_class = None

    @abc.abstractmethod
    def generate_q_expression(self, query):
        """This method should be overridden
        and return a Q() expression."""

    def get(self, request, query):
        try:
            q = self.generate_q_expression(query)
            search = self.document_class.search().query(q)
            response = search.execute()

            print(f'Found {response.hits.total.value} hit(s) for query: "{query}"')

            results = self.paginate_queryset(response, request, view=self)
            serializer = self.serializer_class(results, many=True)
            return self.get_paginated_response(serializer.data)
        except Exception as e:
            return HttpResponse(e, status=500)
```

Notes:

1. To use the class, we have to provide our `serializer_class` and `document_class` and override `generate_q_expression()`.
2. The class does nothing else than run the `generate_q_expression()` query, fetch the response, paginate it, and return serialized data.

All the views should now inherit from `PaginatedElasticSearchAPIView`:

Feedback

```
# search/views.py

import abc

from django.http import HttpResponse
from elasticsearch_dsl import Q
from rest_framework.pagination import LimitOffsetPagination
from rest_framework.views import APIView

from blog.documents import ArticleDocument, UserDocument, CategoryDocument
from blog.serializers import ArticleSerializer, UserSerializer, CategorySerializer


class PaginatedElasticSearchAPIView(APIView, LimitOffsetPagination):
    serializer_class = None
    document_class = None

    @abc.abstractmethod
    def generate_q_expression(self, query):
        """This method should be overridden
        and return a Q() expression."""

    def get(self, request, query):
        try:
            q = self.generate_q_expression(query)
            search = self.document_class.search().query(q)
            response = search.execute()

            print(f'Found {response.hits.total.value} hit(s) for query: "{query}"')

            results = self.paginate_queryset(response, request, view=self)
            serializer = self.serializer_class(results, many=True)
            return self.get_paginated_response(serializer.data)
        except Exception as e:
            return HttpResponse(e, status=500)


# views


class SearchUsers(PaginatedElasticSearchAPIView):
    serializer_class = UserSerializer
    document_class = UserDocument

    def generate_q_expression(self, query):
        return Q('bool',
                 should=[
                     Q('match', username=query),
                     Q('match', first_name=query),
                     Q('match', last_name=query),
                 ], minimum_should_match=1)


class SearchCategories(PaginatedElasticSearchAPIView):
    serializer_class = CategorySerializer
    document_class = CategoryDocument

    def generate_q_expression(self, query):
        return Q(
                'multi_match', query=query,
                fields=[
                    'name',
                    'description',
                ], fuzziness='auto')


class SearchArticles(PaginatedElasticSearchAPIView):
    serializer_class = ArticleSerializer
    document_class = ArticleDocument

    def generate_q_expression(self, query):
        return Q(
                'multi_match', query=query,
                fields=[
                    'title',
                    'author',
                    'type',
```

Feedback

```
                'content'
            ], fuzziness='auto')
```



```
from search.views import SearchArticles, SearchCategories, SearchUsers
```



```
    path('user/<str:query>/', SearchUsers.as_view()),
    path('category/<str:query>/', SearchCategories.as_view()),
    path('article/<str:query>/', SearchArticles.as_view()),
]
```



Then, wire up the app URLs to the project URLs:

```
# core/urls.py

from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('blog/', include('blog.urls')),
    path('search/', include('search.urls')), # new
    path('admin/', admin.site.urls),
]
```

## Testing

Our web application is done. We can test our search endpoints by visiting the following URLs:

| URL | Description |
| --- | --- |
| http://127.0.0.1:8000/search/user/mike/ | Returns user 'mike13' |
| http://127.0.0.1:8000/search/user/jess_/ | Returns user 'jess_' |
| http://127.0.0.1:8000/search/category/seo/ | Returns category 'SEO optimization' |
| http://127.0.0.1:8000/search/category/progreming/ | Returns category 'Programming' |
| http://127.0.0.1:8000/search/article/linux/ | Returns article 'Installing the latest version of Ubuntu' |
| http://127.0.0.1:8000/search/article/java/ | Returns article 'Which programming language is the best?' |

> Notice the typo with the fourth request. We spelled 'progreming', but still got the correct result thanks to fuzziness.

# Alternative Libraries

The path we took isn't the only way to integrate Django with Elasticsearch. There are a few other libraries you *might* want to check out:

1. django-elasicsearch-dsl-drf is a wrapper around Elasticsearch and Django REST Framework. It provides views, serializers, filter backends, pagination and more. It works well, but it might be overkill for smaller projects. I'd

recommend using it if you need advanced Elasticsearch features.

2. Haystack is a wrapper for a number of search backends, like Elasticsearch, Solr, and Whoosh. It allows you to write your search code once and reuse it with different search backends. It works great for implementing a simple search box. Because Haystack is another abstraction layer, there's more overhead involved so you shouldn't use it if performance is really important or if you're working with big amounts of data. It also requires some configuration.
3. Haystack for Django REST Framework is a small library which tries to simplify integration of Haystack with Django REST Framework. At the time of writing, the project is a bit outdated and their documentation is badly written. I've spent a decent amount of time trying to get it to work with no luck.

## Conclusion

In this tutorial, you learned the basics of working with Django REST Framework and Elasticsearch. You now know how to integrate them, create Elasticsearch documents and queries, and serve the data via a RESTful API.

Before launching your project in production, consider using one of the managed Elasticsearch services like Elastic Cloud, Amazon Elasticsearch Service, or Elastic on Azure. The cost of using a managed service will be higher than managing your own cluster, but they provide all of the infrastructure required for deploying, securing, and running Elasticsearch clusters. Plus, they'll handle version updates, regular backups, and scaling.

Grab the code from the django-drf-elasticsearch repo on GitHub.



## Nik Tomazic

Nik is a software developer from Slovenia. He's interested in object-oriented programming and web development. He likes learning new things and accepting new challenges. When he's not coding, Nik's either swimming or watching movies.