The Flask Mega-Tutorial, Part XVI: Full-Text Search

Posted by Miguel Grinberg *on* December 3, 2023 *under* Python Flask Programming Database

This is the sixteenth installment of the Flask Mega-Tutorial series, in which I'm going to add a full-text search capability to Microblog.

You are reading the 2024 edition of the Flask Mega-Tutorial. The complete course is also available to order in e-book and paperback formats from Amazon. Thank you for your support!

If you are looking for the 2018 edition of this course, you can find it here.

For your reference, here is the complete list of articles in this series:

- Chapter 22: Background Jobs
- Chapter 23: Application Programming Interfaces (APIs)

The goal of this chapter is to implement a search feature for Microblog, so that users can find interesting posts using natural language. For many types of websites, it is possible to just let Google, Bing, etc. index all the content and provide search results through their search APIs. This works well for sites that have mostly static pages. But in my application the basic unit of content is a user post, which is a small portion of the entire web page. The type of search results that I want are for these individual blog posts and not entire pages. For example, if I search for the word "dog" I want to see blog posts from any users that include that word. Obviously a page that shows all blog posts that have the word "dog" (or any other possible search term) does not really exist as a page that the big search engines can find and index, so clearly I have no choice other than to roll my own search feature.

The GitHub links for this chapter are: Browse, Zip, Diff.

Introduction to Full-Text Search Engines

Support for full-text search is not standardized like relational databases are. There are several open-source full-text engines: Elasticsearch, Apache Solr, Whoosh, Xapian, Sphinx, etc. As if this isn't enough choice, there are several databases that also provide searching capabilities that are comparable to dedicated search engines like the ones I enumerated above. SQLite, MySQL and PostgreSQL all offer some support for searching text, and NoSQL databases such as MongoDB and CouchDB do too.

If you are wondering which of these can work within a Flask application, the answer is all of them! That is one of the strengths of Flask, it does its job while not being opinionated. So what's the best choice?

From the list of dedicated search engines, Elasticsearch is one that stands out to me as being fairly popular, in part due to its popularity as the "E" in the ELK stack for indexing logs, along with Logstash and Kibana. Using the searching capabilities of one of the relational databases could also be a good choice, but given the fact that SQLAlchemy does not support this functionality, I would have to handle the searching with raw SQL statements, or else find a package that

provides high-level access to text searches while being able to coexist with SQLAlchemy.

Based on the above analysis, I'm going to use Elasticsearch, but I'm going to implement all the text indexing and searching functions in a way that is very easy to switch to another engine. That will allow you to replace my implementation with an alternative one based on a different engine just by rewriting a few functions in a single module.

Installing Elasticsearch

There are several ways to install Elasticsearch, including one-click installers, zip file with the binaries that you need to install yourself, and even a Docker image. The documentation has an Installation page with detailed information on all these options. If you are using Linux, you will likely have a package available for your distribution, but you will need to make sure it is a recent version, ideally 8.0 or newer.

To start a single-node Elasticsearch node for development I suggest the following configuration options:

- `memory="2GB"` to request enough memory for a small index.
- `discovery.type=single-node` to indicate that this is a single-node instead of a cluster.
- `xpack.security.enabled=false` to disable the use of an SSL certificate and user credentials, which are not needed during development.

When running Elasticsearch with Docker, the command to start it is:

```
$ docker run --name elasticsearch -d --rm -p 9200:9200 \
    --memory="2GB" \
    -e discovery.type=single-node -e xpack.security.enabled=false \
    -t docker.elastic.co/elasticsearch/elasticsearch:8.11.1
```

You may need to adjust the version in the command above.

Since I will be managing Elasticsearch from Python, I will also be using the Python client library:

```
(venv) $ pip install elasticsearch
```

You may also want to update your *requirements.txt* file:

```
(venv) $ pip freeze > requirements.txt
```

Elasticsearch Tutorial

I'm going to start by showing you the basics of working with Elasticsearch from a Python shell. This will help you familiarize with this service, so that you can understand the implementation that I will discuss later.

To create a connection to Elasticsearch, create an instance of class `Elasticsearch`, passing a connection URL as an argument:

```
>>> from elasticsearch import Elasticsearch
>>> es = Elasticsearch('http://localhost:9200')
```

Data in Elasticsearch is written to *indexes*. Unlike a relational database, the data is just a JSON object. The following example writes an object with a field called `text` to an index called `my_index`:

```
>>> es.index(index='my_index', id=1, document={'text': 'this is a test
```

For each document stored, Elasticsearch takes a unique id and a dictionary with the data to be stored.

Let's store a second document on this index:

```
>>> es.index(index='my_index', id=2, document={'text': 'a second test'
```

And now that there are two documents in this index, I can issue a free-form search. In this example, I'm going to search for `this test`:

```
>>> es.search(index='my_index', query={'match': {'text': 'this test'}}
```

The response from the `es.search()` call is a response object that wraps a Python dictionary with the search results:

```
ObjectApiResponse({
    'took': 6,
    'timed_out': False,
    '_shards': {'total': 1, 'successful': 1, 'skipped': 0, 'failed': 0
    'hits': {
        'total': {'value': 2, 'relation': 'eq'},
        'max_score': 0.82713,
        'hits': [
            {
                '_index': 'my_index',
                '_id': '1',
                '_score': 0.82713,
                '_source': {'text': 'this is a test'}
            },
            {
                '_index': 'my_index',
                '_id': '2',
                '_score': 0.19363807,
                '_source': {'text': 'a second test'}
            }
        ]
    }
})
```

Here you can see that the search returned the two documents, each with an assigned score. The document with the highest score contains the two words I searched for, and the other document contains only one. You can see that even the best result does not have a perfect score of 1, because the words do not exactly match the text.

Now this is the result if I search for the word `second`:

```
>>> es.search(index='my_index', query={'match': {'text': 'second'}})
ObjectApiResponse({
    'took': 2,
    'timed_out': False,
    '_shards': {'total': 1, 'successful': 1, 'skipped': 0, 'failed': 0
    'hits': {
        'total': {'value': 1, 'relation': 'eq'},
        'max_score': 0.7361701,
        'hits': [
            {
                '_index': 'my_index',
                '_id': '2',
                '_score': 0.7361701,
                '_source': {'text': 'a second test'}
            }
```

```
        ]
    }
})
```

I still get a less than perfect score because my search does not exactly match the text in this document, but since only one of the two documents contains the word "second", the other document does not show up at all.

The Elasticsearch query object has more options, all well documented, and includes options such as pagination and sorting, just like relational databases.

Feel free to add more entries to this index and try different searches. When you are done experimenting, you can delete the index with the following command:

```
>>> es.indices.delete(index='my_index')
```

Elasticsearch Configuration

Integrating Elasticsearch into the application is a great example of the power of Flask. This is a service and Python package combination that does not have anything to do with Flask, yet, I'm going to get a pretty good level of integration, starting from the configuration, which I'm going to write in the `app.config` dictionary from Flask:

config.py: Elasticsearch configuration.

```
class Config:
    # ...
    ELASTICSEARCH_URL = os.environ.get('ELASTICSEARCH_URL')
```

Like with many other configuration entries, the connection URL for Elasticsearch is going to be sourced from an environment variable. If the variable is not defined, I'm going to let the setting be set to `None`, and I'll use that as a signal to disable Elasticsearch. This is mainly for convenience, so that you are not forced to always have the Elasticsearch service up and running when you work on the application, and in particular when you run unit tests. So to make sure the service is used, I

need to define the `ELASTICSEARCH_URL` environment variable, either directly in the terminal or by adding it to the *.env* file as follows:

```
ELASTICSEARCH_URL=http://localhost:9200
```

Elasticsearch presents the challenge that it isn't wrapped by a Flask extension. I cannot create the Elasticsearch instance in the global scope like I did in the examples above because to initialize it I need access to `app.config`, which only becomes available after the `create_app()` function is invoked. My solution is to add a `elasticsearch` attribute to the `app` instance in the application factory function:

app/__init__.py: Elasticsearch instance.

```
# ...
from elasticsearch import Elasticsearch

# ...

def create_app(config_class=Config):
    app = Flask(__name__)
    app.config.from_object(config_class)

    # ...
    app.elasticsearch = Elasticsearch([app.config['ELASTICSEARCH_URL']]
        if app.config['ELASTICSEARCH_URL'] else None

    # ...
```

Adding a new attribute to the `app` instance may seem a little strange, but Python objects are not strict in their structure, and new attributes can be added to them at any time. An alternative that you may also consider is to create a subclass of `Flask` (maybe called `Microblog`), with the `elasticsearch` attribute defined in its `__init__()` function.

Note how I use a conditional expression to make the Elasticsearch instance `None` when a URL for the Elasticsearch service wasn't defined in the environment.

A Full-Text Search Abstraction

As I said in the chapter's introduction, I want to make it easy to switch from Elasticsearch to other search engines, and I also don't want to code

this feature specifically for searching blog posts, I prefer to design a solution that in the future I can easily extend to other models if I need to. For all these reasons, I decided to create an *abstraction* for the search functionality. The idea is to design the feature in generic terms, so I will not be assuming that the `Post` model is the only one that needs to be indexed, and I will also not be assuming that Elasticsearch is the index engine of choice. But if I can't make any assumptions about anything, how can I get this work done?

The first thing that I need, is to somehow find a generic way to indicate which model and which field or fields in it are to be indexed. I'm going to say that any model that needs indexing needs to define a `__searchable__` class attribute that lists the fields that need to be included in the index. For the `Post` model, these are the changes:

app/models.py: Add a **searchable** attribute to the Post model.

```
class Post(db.Model):
    __searchable__ = ['body']
    # ...
```

So here I'm saying that this model needs to have its `body` field indexed for searching. But just to make sure this is perfectly clear, this `__searchable__` attribute that I added is just a variable, it does not have any behavior associated with it. It will just help me write my indexing functions in a generic way.

I'm going to write all the code that interacts with the Elasticsearch index in a *app/search.py* module. The idea is to keep all the Elasticsearch code in this module. The rest of the application will use the functions in this new module to access the index and will not have direct access to Elasticsearch. This is important, because if one day I decided I don't like Elasticsearch anymore and want to switch to a different engine, all I need to do is rewrite the functions in this module, and the application will continue to work as before.

For this application, I decided that I need three supporting functions related to text indexing: I need to add entries to a full-text index, I need to remove entries from the index (assuming one day I will support deleting blog posts), and I need to execute a search query. Here is the *app/search.py* module that implements these three functions for

Elasticsearch, using the functionality I showed you above from the Python console:

app/search.py: Search functions.

```
from flask import current_app

def add_to_index(index, model):
    if not current_app.elasticsearch:
        return
    payload = {}
    for field in model.__searchable__:
        payload[field] = getattr(model, field)
    current_app.elasticsearch.index(index=index, id=model.id, document

def remove_from_index(index, model):
    if not current_app.elasticsearch:
        return
    current_app.elasticsearch.delete(index=index, id=model.id)

def query_index(index, query, page, per_page):
    if not current_app.elasticsearch:
        return [], 0
    search = current_app.elasticsearch.search(
        index=index,
        query={'multi_match': {'query': query, 'fields': ['*']}},
        from_=(page - 1) * per_page,
        size=per_page)
    ids = [int(hit['_id']) for hit in search['hits']['hits']]
    return ids, search['hits']['total']['value']
```

These functions all start by checking if `app.elasticsearch` is `None`, and in that case return without doing anything. This is so that when the Elasticsearch server isn't configured, the application continues to run without the search capability and without giving any errors. This is just as a matter of convenience during development or when running unit tests.

The functions accept the index name as an argument. The functions that add and remove entries from the index take the SQLAlchemy model as a second argument. The `add_to_index()` function uses the `__searchable__` class variable I added to the model to build the document that is inserted into the index. If you recall, Elasticsearch documents also needed a unique identifier. For that I'm using the `id` field of the SQLAlchemy model, which is also conveniently unique. Using the same `id` value for SQLAlchemy and Elasticsearch is very useful

when running the searches, as it allows me to link entries in the two databases. Something I did not mention above is that if you attempt to add an entry with an existing `id`, then Elasticsearch replaces the old entry with the new one, so `add_to_index()` can be used for new objects as well as for modified ones.

I did not show you the `es.delete()` function that I'm using in `remove_from_index()` before. This function deletes the document stored under the given `id`. Here is a good example of the convenience of using the same `id` to link entries in both databases.

The `query_index()` function takes the index name and a text to search for, along with pagination controls, so that search results can be paginated like Flask-SQLAlchemy results are. You have already seen an example usage of the `es.search()` function from the Python console. The call I'm issuing here is fairly similar, but instead of using a `match` query type, I decided to use `multi_match`, which can search across multiple fields. By passing a field name of `*`, I'm telling Elasticsearch to look in all the fields that are indexed, so basically I'm searching the entire index. This is useful to make this function generic, since different models can have different field names in the index.

The `from_` and `size` arguments control what subset of the entire result set needs to be returned. Elasticsearch does not provide a nice `Pagination` object like the one from Flask-SQLAlchemy, so I have to do the pagination math to calculate the `from` value.

The `return` statement in the `query_index()` function is somewhat complex. It returns two values: the first is a list of `id` elements for the search results, and the second returned value is the total number of results. Both are obtained from the Python dictionary returned by the `es.search()` function. If you are not familiar with the expression that I'm using to obtain the list of IDs, this is called a *list comprehension*, and is a fantastic feature of the Python language that allows you to transform lists from one format to another. In this case I'm using the list comprehension to extract the `id` values from the much larger list of results provided by Elasticsearch.

Is this too confusing? Maybe a demonstration of these functions from the Python console can help you understand them a bit more. In the following session, I manually add all the posts from the database to the

Elasticsearch index. In my development database, I wrote a few posts that had the numbers "one", "two", "three", "four" and "five" in them, so I used that as a search query. You may need to adapt your query to match the contents of your database:

```
>>> from app.search import add_to_index, remove_from_index, query_inde
>>> for post in db.session.scalars(sa.select(Post)):
...     add_to_index('posts', post)
>>> query_index('posts', 'one two three four five', 1, 100)
([15, 13, 12, 4, 11, 8, 14], 7)
>>> query_index('posts', 'one two three four five', 1, 3)
([15, 13, 12], 7)
>>> query_index('posts', 'one two three four five', 2, 3)
([4, 11, 8], 7)
>>> query_index('posts', 'one two three four five', 3, 3)
([14], 7)
```

The query that I issued returned seven results. When I asked for page 1 with 100 items per page I get all seven, but then the next three examples shows how I can paginate the results in a way that is very similar to what I did for Flask-SQLAlchemy, with the exception that the results come as a list of IDs instead of SQLAlchemy objects.

If you want to keep things clean, delete the `posts` index after you are doing experimenting with it:

```
>>> app.elasticsearch.indices.delete(index='posts')
```

Integrating Searches with SQLAlchemy

The solution that I showed you in the previous section is decent, but it still has a couple of problems. The most obvious problem is that results come as a list of numeric IDs. This is highly inconvenient, I need SQLAlchemy models so that I can pass them down to templates for rendering, and I need a way to replace that list of numbers with the corresponding models from the database. The second problem is that this solution requires the application to explicitly issue indexing calls as posts are added or removed, which is not terrible, but less than ideal, since a bug that causes a missed indexing call when making a change on the SQLAlchemy side is not going to be easily detected, the two databases will get out of sync more and more each time the bug occurs and you will probably not notice for a while. A better solution would be

for these calls to be triggered automatically as changes are made on the SQLAlchemy database.

The problem of replacing the IDs with objects can be addressed by creating a SQLAlchemy query that reads those objects from the database. This sounds easy in practice, but doing it efficiently with a single query is actually a bit tricky to implement.

For the problem of triggering the indexing changes automatically, I decided to drive updates to the Elasticsearch index from SQLAlchemy *events*. SQLAlchemy provides a large list of events that applications can be notified about. For example, each time a session is committed, I can have a function in the application invoked by SQLAlchemy, and in that function I can apply the same updates that were made on the SQLAlchemy session to the Elasticsearch index.

To implement the solutions to these two problems I'm going to write a *mixin* class. Remember mixin classes? In Chapter 5, I added the `UserMixin` class from Flask-Login to the `User` model, to give it some features that were required by Flask-Login. For the search support I'm going to define my own `SearchableMixin` class, that when attached to a model, will give it the ability to automatically manage an associated full-text index. The mixin class will act as a "glue" layer between the SQLAlchemy and Elasticsearch worlds, providing solutions to the two problems I stated above.

Let me show you the implementation, then I'll go over some interesting details. Note that this makes use of several advanced techniques, so you will need to study this code carefully to fully understand it.

app/models.py: SearchableMixin class.

```
from app.search import add_to_index, remove_from_index, query_index

class SearchableMixin(object):
    @classmethod
    def search(cls, expression, page, per_page):
        ids, total = query_index(cls.__tablename__, expression, page,
        if total == 0:
            return [], 0
        when = []
        for i in range(len(ids)):
            when.append((ids[i], i))
        query = sa.select(cls).where(cls.id.in_(ids)).order_by(
            db.case(*when, value=cls.id))
```

```
        return db.session.scalars(query), total

    @classmethod
    def before_commit(cls, session):
        session._changes = {
            'add': list(session.new),
            'update': list(session.dirty),
            'delete': list(session.deleted)
        }

    @classmethod
    def after_commit(cls, session):
        for obj in session._changes['add']:
            if isinstance(obj, SearchableMixin):
                add_to_index(obj.__tablename__, obj)
        for obj in session._changes['update']:
            if isinstance(obj, SearchableMixin):
                add_to_index(obj.__tablename__, obj)
        for obj in session._changes['delete']:
            if isinstance(obj, SearchableMixin):
                remove_from_index(obj.__tablename__, obj)
        session._changes = None

    @classmethod
    def reindex(cls):
        for obj in db.session.scalars(sa.select(cls)):
            add_to_index(cls.__tablename__, obj)

db.event.listen(db.session, 'before_commit', SearchableMixin.before_co
db.event.listen(db.session, 'after_commit', SearchableMixin.after_comm
```

There are four functions in this mixin class, all class methods. Just as a refresher, a class method is a special method that is associated with the class and not a particular instance. Note how I renamed the `self` argument used in regular instance methods to `cls`, to make it clear that this method receives a class and not an instance as its first argument. Once attached to the `Post` model for example, the `search()` method above would be invoked as `Post.search()`, without having to have an actual instance of class `Post`.

The `search()` class method wraps the `query_index()` function from *app/search.py* to replace the list of object IDs with actual objects from SQLAlchemy. You can see that the first thing this function does is call `query_index()`, passing `cls.__tablename__` as the index name. This is going to be a convention, all indexes will be named with the name Flask-SQLAlchemy assigned to the relational table. The function returns the

list of result IDs, and the total number of results. The SQLAlchemy query that retrieves the list of objects by their IDs is based on a `CASE` statement from the SQL language, which needs to be used to ensure that the results from the database come in the same order as the IDs returned by Elasticsearch, which are sorted by relevance. If you want to learn more about the way this query works, you can consult the accepted answer to this StackOverflow question. The `search()` function returns the results from the query that replaces the list of IDs with objects, and also passes through the total number of search results as a second return value.

The `before_commit()` and `after_commit()` methods are going to respond to two events from SQLAlchemy, which are triggered before and after a commit takes place respectively. The before handler is useful because the session hasn't been committed yet, so I can look at it and figure out what objects are going to be added, modified and deleted, available as `session.new`, `session.dirty` and `session.deleted` respectively. These objects are not going to be available anymore after the session is committed, so I need to save them before the commit takes place. I'm using a `session._changes` dictionary to write these objects in a place that is going to survive the session commit, because as soon as the session is committed I will be using them to update the Elasticsearch index.

When the `after_commit()` handler is invoked, the session has been successfully committed, so this is the proper time to make changes on the Elasticsearch side. The session object has the `_changes` variable that I added in `before_commit()`, so now I can iterate over the added, modified and deleted objects, and make the corresponding calls to the indexing functions in *app/search.py* for the objects that have the `SearchableMixin` class.

The `reindex()` class method is a simple helper method that you can use to refresh an index with all the data from the relational side. You saw me do something similar from the Python shell session above to do an initial load of all the posts into a test index. With this method in place, I can issue `Post.reindex()` to add all the posts in the database to the search index.

After the class definition I made two calls to SQLALchemy's function `db.event.listen()`. Note that these calls are not inside the class, but

after it. The purpose of these two statements is to set up the event handlers that will make SQLAlchemy call the `before_commit()` and `after_commit()` methods before and after each commit respectively.

To incorporate the `SearchableMixin` class into the `Post` model I have to add it as a subclass, and I also need to hook up the before and after commit events:

app/models.py: Adding the SearchableMixin class to the Post model.

```
class Post(SearchableMixin, db.Model):
    # ...
```

Now the `Post` model is automatically maintaining a full-text search index for posts. I can use the `reindex()` method to initialize the index from all the posts currently in the database:

```
>>> Post.reindex()
```

And I can search posts working with SQLAlchemy models by running `Post.search()`. In the following example, I ask for the first page of five elements for my query:

```
>>> query, total = Post.search('one two three four five', 1, 5)
>>> total
7
>>> query.all()
[<Post five>, <Post two>, <Post one>, <Post one more>, <Post one>]
```

Search Form

This was very intense. The work that I've done above to keep things generic touches on several advanced topics, so it may take you time to fully understand it. But now I have a complete system to work with natural language searches for blog posts. What I need to do now is integrate all this functionality with the application.

A fairly standard approach for web-based searches is to have the search term as a `q` argument in the query string of the URL. For example, if you wanted to search for `Python` on Google, and you want to save a couple of seconds, you can just type the following URL in your browser's address bar to go directly to the results:

```
https://www.google.com/search?q=python
```

Allowing searches to be completely encapsulated in a URL is nice, because these can be shared with other people, who just by clicking on the link have access to the search results.

This introduces a change in the way I showed you to handle web forms in the past. I have used `POST` requests to submit form data for all the forms the application has so far, but to implement searches as above, the form submission will have to go as a `GET` request, which is the request method that is used when you type a URL in your browser or click a link. Another interesting difference is that the search form is going to be in the navigation bar, so it needs to be present in all pages of the application.

Here is the search form class, with just the `q` text field:

app/main/forms.py: Search form.

```
from flask import request

class SearchForm(FlaskForm):
    q = StringField(_l('Search'), validators=[DataRequired()])

    def __init__(self, *args, **kwargs):
        if 'formdata' not in kwargs:
            kwargs['formdata'] = request.args
        if 'meta' not in kwargs:
            kwargs['meta'] = {'csrf': False}
        super(SearchForm, self).__init__(*args, **kwargs)
```

The `q` field does not require any explanation, as it is similar to other text fields I've used in the past. For this form, I decided not to have a submit button. For a form that has a text field, the browser will submit the form when you press Enter with the focus on the field, so a button is not needed. I have also added a `__init__` constructor function, which provides values for the `formdata` and `meta` arguments if they are not provided by the caller. The `formdata` argument determines from where Flask-WTF gets form submissions. The default is to use `request.form`, which is where Flask puts form values that are submitted via `POST` request. Forms that are submitted via `GET` request have the field values in the query string, so I need to point Flask-WTF at `request.args`, which is where Flask writes the query string arguments. And as you remember,

forms have CSRF protection added by default, with the inclusion of a CSRF token that is added to the form via the `form.hidden_tag()` construct in templates. For clickable search links to work, CSRF needs to be disabled, so I'm setting `meta` to `{'csrf': False}` so that Flask-WTF knows that it needs to bypass CSRF validation for this form.

Since I'm going to need to have this form visible in all pages, I need to create an instance of the `SearchForm` class regardless of the page the user is viewing. The only requirement is that the user is logged in, because for anonymous users I am not currently showing any content. Instead of creating a form object in every route, and then passing the form to all the templates, I'm going to show you a very useful trick that eliminates duplication of code when you need to implement a feature across the entire application. I already used a `before_request` handler before, back in Chapter 6, to record the time of last visit for each user. What I'm going to do is create my search form in that same function, but with a twist:

app/main/routes.py: Instantiate the search form in the before_request handler.

```
from flask import g
from app.main.forms import SearchForm

@bp.before_app_request
def before_request():
    if current_user.is_authenticated:
        current_user.last_seen = datetime.now(timezone.utc)
        db.session.commit()
        g.search_form = SearchForm()
    g.locale = str(get_locale())
```

Here I create an instance of the search form class when I have an authenticated user. But of course, I need this form object to persist until it can be rendered at the end of the request, so I need to store it somewhere. That somewhere is going to be the `g` container, provided by Flask. This `g` variable provided by Flask is a place where the application can store data that needs to persist through the life of a request. Here I'm storing the form in `g.search_form`, so then when the before request handler ends and Flask invokes the view function that handles the requested URL, the `g` object is going to be the same, and will still have the form attached to it. It's important to note that this `g` variable is specific to each request and each client, so even if your web server is handling multiple requests at a time for different clients, you can still rely

on `g` to work as private storage for each request, independently of what goes on in other requests that are handled concurrently.

The next step is to render the form to the page. I said above that I wanted this form in all pages, so what makes more sense is to render it as part of the navigation bar. This is, in fact, simple, because templates can also see the data stored in the `g` variable, so I don't need to worry about adding the form as an explicit template argument in all the `render_template()` calls in the application. Here is how I can render the form in the base template:

app/templates/base.html: Render the search form in the navigation bar.

```
            ...
            <div class="collapse navbar-collapse" id="navbarSupportedC
                <ul class="navbar-nav me-auto mb-2 mb-lg-0">
                    ... home and explore links ...
                </ul>
                {% if g.search_form %}
                <form class="navbar-form navbar-left" method="get"
                        action="{{ url_for('main.search') }}">
                    <div class="form-group">
                        {{ g.search_form.q(size=20, class='form-contro
                            placeholder=g.search_form.q.label.text) }}
                    </div>
                </form>
                {% endif %}
                ...
```

The form is rendered only if `g.search_form` is defined. This check is necessary because some pages, such as error pages, may not have it defined. This form is slightly different from the ones I did previously. I'm setting its `method` attribute to `get`, because I want the form data to be submitted on the query string with a `GET` request. Also, the other forms I created had the `action` attribute empty, because they were submitted to the same page that rendered the form. This form is special because it appears in all pages, so I need to tell it explicitly where it needs to be submitted, which is a new route that is specifically dedicated to handling searches.

Search View Function

The last bit of functionality to complete the search feature is the view function that receives the search form submission. This view function is going to be attached to the */search* route, so that you can send a search request with a *http://localhost:5000/search?q=search-words*, just like Google.

app/main/routes.py: Search view function.

```
@bp.route('/search')
@login_required
def search():
    if not g.search_form.validate():
        return redirect(url_for('main.explore'))
    page = request.args.get('page', 1, type=int)
    posts, total = Post.search(g.search_form.q.data, page,
                               current_app.config['POSTS_PER_PAGE'])
    next_url = url_for('main.search', q=g.search_form.q.data, page=pag
        if total > page * current_app.config['POSTS_PER_PAGE'] else No
    prev_url = url_for('main.search', q=g.search_form.q.data, page=pag
        if page > 1 else None
    return render_template('search.html', title=_('Search'), posts=pos
                           next_url=next_url, prev_url=prev_url)
```

You have seen that in the other forms I used the `form.validate_on_submit()` method to check if the form submission was valid. Unfortunately that method only works for forms submitted via `POST` request, so for this form I need to use `form.validate()` which just validates field values, without checking how the data was submitted. If the validation fails, it is because the user submitted an empty search form, so in that case I just redirect to the explore page, which shows all blog posts.

The `Post.search()` method from my `SearchableMixin` class is used to obtain the list of search results. The pagination is handled in a very similar way to that of the index and explore pages, but generating the next and previous links is a bit trickier without the help of the `Pagination` object from Flask-SQLAlchemy. This is where the total number of results passed as a second return value from `Post.search()` is useful.

Once the page of search results and pagination links are calculated, all that is left is to render a template with all this data. I could have figured out a way to reuse the *index.html* template to display search results, but given that there are a few differences I decided to create a dedicated

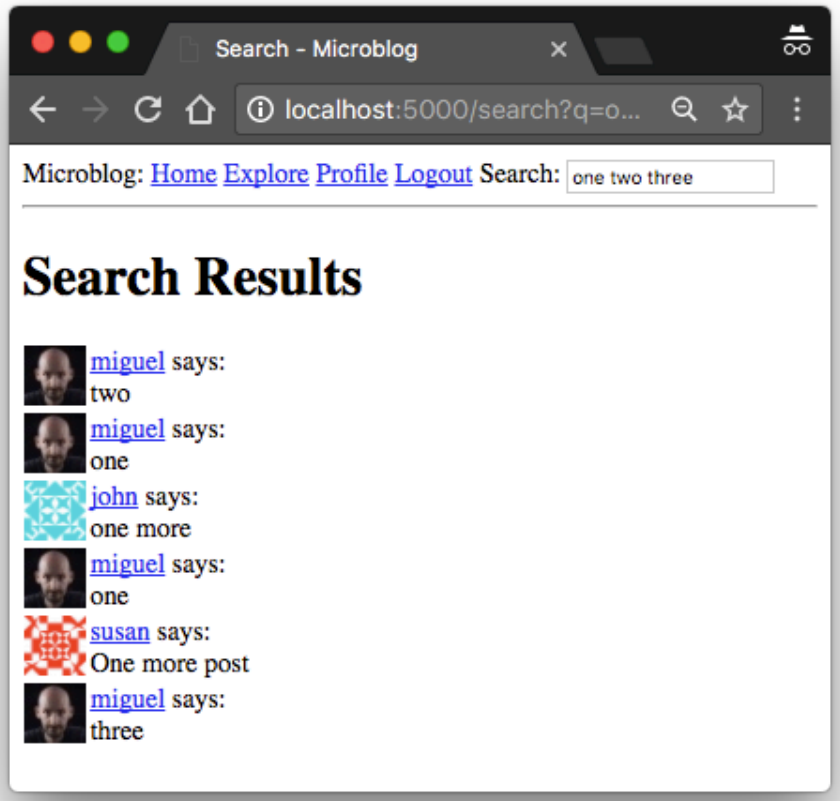
search.html template that is dedicated to showing search results, taking advantage of the *_post.html* sub-template to render the search results:

app/templates/search.html: Search results template.

```
{% extends "base.html" %}

{% block content %}
    <h1>{{ _('Search Results') }}</h1>
    {% for post in posts %}
        {% include '_post.html' %}
    {% endfor %}
    <nav aria-label="Post navigation">
        <ul class="pagination">
            <li class="page-item{% if not prev_url %} disabled{% endif
                <a class="page-link" href="{{ prev_url }}">
                    <span aria-hidden="true">&larr;</span> {{ _('Newer
                </a>
            </li>
            <li class="page-item{% if not next_url %} disabled{% endif
                <a class="page-link" href="{{ next_url }}">
                    {{ _('Older posts') }} <span aria-hidden="true">&r
                </a>
            </li>
        </ul>
    </nav>
{% endblock %}
```

The rendering logic for the previous and next links is similar to what I used in the *index.html* and *user.html* templates.



What do you think? This was an intense chapter, where I presented some fairly advanced techniques. Some concepts in this chapter may take some time to sink in. The most important take away from this chapter is that if you want to use a different search engine than Elasticsearch, all you need to do is re-implement the three functions in *app/search.py*. The other important benefit of going through this effort is that in the future, if I need to add search support for a different database model, I can simply do so by adding the `SearchableMixin` class to it, the `__searchable__` attribute with the list of fields to index and the SQLAlchemy event handler connections. I think it was well worth the effort, because from now on, it is going to be easy to deal with full-text indexes.

Continue on to the next chapter.

Buy me a coffee?

Thank you for visiting my blog! If you enjoyed this article, please consider supporting my work and keeping me caffeinated with a small one-time donation through Buy me a coffee. Thanks!

Share this post

23 comments

#1 **Vladimir Kuzmenkov** said a year ago

Intense chapter indeed. Way too advanced to wrap one's head around it from the first time for sure.

Would like to share that upon downloading elasticsearch one might want to amend elasticsearch.yml config file and set xpack.security.enabled to false in order to access via http://localhost:9200. Otherwise elasticserarch service available only via https and requires credentials.

Thanks!

#2 Miguel Grinberg said a year ago

@Vladimir: you can disable security in the command that starts the Docker container, as shown above in this article.

#3 **Leo** said a year ago

since you've only allowed g.search_form to be created if user authed, the login page fails since a user is not authed but the base template still tries to find g.search_form

#4 Miguel Grinberg said a year ago

@Leo: if you have a specific error to share, please do so. The base template should not have any issues because it checks that `g.search_form` exists before rendering the search form.

#5 **David** said a year ago

Hi Miguel, thanks a lot for this great tutorial. I'm applying all the techniques you showed in this post on my own project.

In my case, inside the function "add_to_index", it needs to retrieve a relationship model to get all the data for the payload. And I use "SearchableMixin" the same way you showed here. Now, I've been getting this error each time I try to call db.session.commit() after updating a model record. Here is the partial error message:

"sqlalchemy.exc.InvalidRequestError: This session is in 'committed' state; no further SQL can be emitted within this transaction."

The error occurs during calling add_to_index() inside the after_commit() method for obj stored in session._changes['update']. Is there any way I can avoid this error? Thanks.

#6 Miguel Grinberg said a year ago

@David: the solution that I present in this chapter integrates an Elasticsearch index with one (only one) database table. It is not designed to have an ES index that maps to multiple database tables. I suggest if you need something like that you implement separate functions to maintain the index, because the one-to-one mapping between database and ES is key to this solution.

#7 **Yuming** said a year ago

Hi Miguel,
This is a follow up to my comment #389 in Chapter 16 of your 2018 tutorial at https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-xvi-full-text-search-2018/page/16#comments

In that comment I reported a problem with using Elasticsearch (ES) with Chinese language: While ES uses English words as “search terms”, it treats Chinese characters as search terms because Chinese words are composed of 2 ~ 4 characters and each character has its own definite and rich meaning (which is different from English syllables). For example, if I search for Chinese word “牛肉” (beef), which is composed of two characters 牛(cow) and 肉(meat), documents containing only 牛 or 肉 will also be reported as hits and the documents that have both characters may have lower scores if at least one of the two characters is used frequently by other words. This problem makes ES almost useless for searching for Chinese words.

I came up with an idea: To allow users to search for a Chinese word instead of its characters, my code searches for all the characters of the word one by one. Searching for each character returns a hit list, then I merge the hit lists of all the characters into a hit list for the word, which contains only the IDs of the documents that use ALL of the characters. In other words, although ES considers documents that contain any character used by the word as hits, my code considers only the documents that contain all the characters of the word as hits.

This method has a problem if I search for Chinese words using ES pagination: If the document IDs reported by ES based on character searching appear in different pages, then my algorithm will fail to merge them into a merged list for the word. My solution is to use a page size that is guaranteed to be larger than the total number of documents my app will ever have. In other words, I make sure that the first page of ES’s pagination is larger enough to carry the searching result.

How do I display search results nicely with a huge ES pagination if the hit list is very long? My solution is to do my own pagination with Javascript at the frontend.

This method can be used to search not only for Chinese words, but also for Chinese sentences. With about 60 documents currently in my ES index and searching for some long Chinese sentences, I don’t see any performance problems due to the large ES pagination page size. You can try it at www.chinesesong.net if interested, this app implements almost all the advanced features covered by your tutorial plus some other features such as AWS lambda, Youtube API, etc.

Of course, if an ES index has more than a few thousands of documents, this method will probably cause search performance to drop, so this is not a generic solution, but it works for apps like mine, which has only a limited number of Chinese documents in the ES index.

#8 Miguel Grinberg said a year ago

@Yuming: The problem here is that the default method in which Elasticsearch processes the text is not appropriate for Chinese and other Asian languages. But you can configure your index to analyze text using a better method. See this blog post.

#9 **Yuming** said a year ago

Hi Miguel,
Thank you for the information about the Elasticsearch (ES) plugins for handling some East Asian languages. Obviously the plugins try to solve the problem that I encountered when using ES for Chinese documents. I guess the plugins are able to “analyze” Chinese documents and figure out if a sequence of 2~4 characters should be treated as independent characters or a Chinese word (Chinese characters can be as meaningful as words), then those words will be used by ES as "search terms" in addition to characters.

It is interesting that the author of the blog said that the standard ES analyzer still has problems processing some nouns that are not words (the author used “冬季奥运会“, which means “Winter Olympics”, as an example), so in addition to the standard analyzer, it may also be necessary to install language specific plugins.

My approach is not generic, but seems to be able to solve the problem without those plugins for apps that will never use more than a limited amount of data in its ES index.

#10 **Noah** said a year ago

Hi, Miguel.

> *a bug that causes a missed indexing call when making a change on the SQLAlchemy side is not going to be easily detected*

Could you elaborate this bug? I don't understand which scenarios will cause this.

#11 **Quy** said a year ago

Thanks for the great tutorial Miguel!

I'm just wondering is it a good idea to get this search functionality be a module/blueprint of its own? Was there any design decision you have made about this matter?

#12 Miguel Grinberg said a year ago

@Noah: this would happen when you add new code that adds blog posts to the database and you forget that you also need to add the code updates the full-text search index with the same change.

#13 Miguel Grinberg said a year ago

@Quy: The search functionality is already in a module of its own called `search.py`.

#14 **Amos** said a year ago

Hi Miguel. Thanks for the great tutorial. I am getting the error below while trying to implement the search functionality.
elasticsearch.NotFoundError: NotFoundError(404, '<!doctype html>\n<html lang=en>\n<title>404 Not Found</title>\n<h1>Not Found</h1>\n

The requested URL was not found on the server. If you entered the URL manually please check your spelling and try again.

\n')
I am wondering if the issue stems around this part of the

route.py:
@bp.route('/search')
@login_required
def search():
.....
next_url = url_for('main.search', q=g.search_form.q.data, page=page + 1) \
Is it okay for one to define a route for the first time as @bp.route('/search') and apply the same route within the body of its definition function as url_for('main.search'). I am thinking that the function is searching for the 'main.search' route which is in the process of being defined.

#15 Miguel Grinberg said a year ago

@Amos: please compare your code against mine carefully, you must have a mistake that is causing this.

#16 **joshua** said 9 months ago

Ever worked with typesense.Can we get your guide on using it for full text search as alternative to elastic search.

#17 Miguel Grinberg said 9 months ago

@joshua: I explain in this article how you can replace Elasticsearch with the search engine of your choice. I can't provide the code for every single full-text search engine out there.

#18 **Dev** said 7 months ago

I have been following your tutorial for a while now, and I gotta say... This is one of the hardest lessons in this tutorial another one being the followers chapter. But all in all I'm learning a lot from these chapters. So thank you!

#19 **DS** said 4 months ago

Hi Miguel, this tutorial has taught me a lot so far, so thank you! I made it to this chapter and everything works, except for two things:

Email - because my SMTP settings are not in agreement apparently, but I will check this later.
Ajax translation - because I don't have a credit card and cannot sign up for Azure or Google Cloud.

I can live without those functionalities for now, but getting to this chapter, I feel like I'm missing something. I'm creating my app on PythonAnywhere and syncing it with GitHub, but now all of a sudden it seems like I need Docker, so do I need to move the whole project to my laptop? I want to prevent not being able to have my app run on an actual live server.

#20 Miguel Grinberg said 4 months ago

@DS: Docker is not required, it is just the most convenient when you are developing the application locally, which is what I had in mind when I designed this tutorial. Another option that you have is to get an Elastic cloud account (free for 14 days only, sadly) and host your Elasticsearch database there. Aside from slightly more complex connection arguments everything else related to Elasticsearch should work the same with a cloud hosted database.

#21 **DS** said 4 months ago

Hi Miguel, thanks for the quick reply! In the meantime I managed to make email work through SendGrid's API - I was probably running into another limitation of PythonAnywhere with my SMTP settings. They also don't seem to support ElasticSearch, so for now I think I will skip this chapter. I imagine I can always come back to it :)

#22 **BumblyCodesNowIGuess** said 2 months ago

Hi Miguel,

Thank you for this great tutorial. I am doing a proof of concept for a project but am using flask, sqlalchemy and wtforms to

create the search functionality, and will not have an ES instance. So far I have gotten it to return me results perfectly, but I am struggling to make the results 'clickable', to navigate to the user profile. Can you help me out a little bit?

I was following along but can't get my head around how Flask recognizes and generates the URL to point to the user profile. I'll post what I have:

---------routes.py--------

```
...
@app.route('/search', methods=['GET', 'POST'])
def search():
    form = SearchForm()
    results = []
    if form.validate_on_submit():
        query = form.search_query.data
        results = db.session.query(User).filter(User.r
    return render_template('search.html', form=form, r
....
```

--------forms.py----------

```
...
class SearchForm(FlaskForm):
    search_query = StringField('Name', validators=[Dat

    def __init__(self, *args, **kwargs):
        if 'formdata' not in kwargs:
            kwargs['formdata'] = request.args
        if 'meta' not in kwargs:
            kwargs['meta'] = {'csrf': False}
        super(SearchForm, self).__init__(*args, **kwar
...
```

---------search.html----------

```
{% extends "base.html" %}

{% block content %}
    <title>Search</title>
    <h2>Search for User</h2>
    <form method="POST">
        {{ form.hidden_tag()}}
        {{ form.search_query.label }} {{ form.search_q
```

```
        <button type="submit">Search</button>
    </form>
    <h2> Results </h2>
    <ul>
        {% for result in results %}
        <li>ID: {{ result.id }}, Name: {{ result.name
        {% else %}
        <li> No Results Found.</li>
        {% endfor %}
    </ul>
{% endblock %}
```

TYIA for anything you can provide to help. This is my first time working with Flask and your book has become a literal bible for me to reference on a minute-by-minute basis.

#23 Miguel Grinberg said 2 months ago

@BumblyCodesNowIGuess: your link should be something like this:

```
<li><a href="{{ url_for('user', id=result.id) }}">{{ .
```

You may need to adjust the endpoint name and the endpoint arguments to match your application.

«« « » »»

Leave a Comment

Name

Email

Comment

Captcha

I'm not a robot

reCAPTCHA
Privacy - Terms

Submit

The Flask Mega-Tutorial

New 2024 Edition!

If you would you like to support my work on my Flask Mega-Tutorial series on this blog and as a reward have access to the complete tutorial nicely structured as a book and/or a set of videos, you can now order it from my Courses site or from Amazon.

Click here to get the Book!
Click here to get the Video Course!

About Miguel

Welcome to my blog!

I'm a software engineer and technical writer, currently living in Drogheda, Ireland.

You can also find me on Github, LinkedIn, Bluesky, Mastodon, Twitter, YouTube, and Patreon.

Thank you for visiting!

Categories

- AI *3*
- Arduino *7*
- Authentication *10*
- Blog *1*
- C++ *5*
- CSS *1*
- Cloud *11*
- Database *23*
- Docker *5*
- Filmmaking *6*
- Flask *129*
- Games *1*
- IoT *8*
- JavaScript *36*
- MicroPython *9*
- Microdot *1*
- Microservices *2*
- Movie Reviews *5*
- Personal *3*
- Photography *7*
- Product Reviews *2*
- Programming *192*
- Project Management *1*
- Python *174*
- REST *7*
- Raspberry Pi *8*
- React *19*
- Reviews *1*

- Robotics 6
- Security 12
- Video 22
- WebSocket 2
- Webcast 3
- Windows 1