Django-SphinxQL Documentation

Release 0.1

jorgecarleitao

Sep 27, 2017

Contents

Django-SphinxQL is an API to use sphinx search in Django. Thanks for checking it out.

Django is a Web framework for building websites with relational databases; Sphinx is a search engine designed for relational databases. Django-SphinxQL defines an ORM for using **Sphinx in Django**. As corollary, it allows you to implement full text search with Sphinx in your Django website.

Specifically, this API allows you to:

1. Configure Sphinx with Python.
2. Index Django models in Sphinx.
3. Execute Sphinx queries (SphinxQL) and have the results in Django.

Check README for a quick overview of how to quickly get it working. Check this documentation for a detailed explanation of everything that is available in Django-Sphinxql.

CHAPTER 1

Indexes API

This document explains how Django-SphinxQL maps a Django model into a Sphinx index.

Sphinx uses a SQL query to retrieve data from a relational database to index it. This means that Django-SphinxQL must know:

1. what you want to index (e.g. what data)
2. how you want to index it (e.g. type)

In the same spirit of Django, Django-SphinxQL defines an ORM for you to answer those questions. For example:

```python
# indexes.py
from sphinxql import fields, indexes
from myapp import models

class PostIndex(indexes.Index):
    text = fields.Text('text')  # Post has the model field ``text``
    date = fields.Date('added_date')
    summary = fields.Text('summary')
    # `Post` has a foreign key to a `Blog`, and blog has a name.
    blog_name = fields.Text('blog__name')

    class Meta:
        model = models.Post  # the model we are indexing
```

The `fields` and the `Meta.model` identify the "what"; the specific field type, e.g. `Text`, identifies the "how". In the following sections the complete API is presented.

API references

Index

class `indexes.`**`Index`**

`Index` is an ORM to Sphinx-index a Django Model. It works in a similar fashion to a Django model: you set up the *`fields`*, and it constructs a Sphinx index out of those fields.

Formally, when an index is declared, it is registered in the *`IndexConfigurator`* so Django-SphinxQL configures it in Sphinx.

An index is always composed by two components: a set of *`fields`* that you declare as class attributes and a class `Meta`:

class `Meta`

Used to declare Django-SphinxQL related options. An index must always define the `model` of its Meta:

`model`

The model of this index. E.g. `model = blog.models.Post`.

In case you want to index only particular instances, you can define the class attribute `query`:

`query`

Optional. The query Sphinx uses to index its data, e.g. `query = models.Post.objects.filter(date__year__gt=2000)`. If not set, Django-SphinxQL uses `.objects.all()`. This is useful if you want to construct indexes for specific sets of instances.

`range_step`

Optional. Defining it automatically enables ranged-queries. This integer defines the number of rows per query retrieved during indexing. It increases the number of queries during indexing, but reduces the amount of data transfer on each query.

In case you want to override Sphinx settings only to this particular index, you can also define the following class attributes:

`source_params`

A dictionary of Sphinx options to override Sphinx settings of `source` for this particular index.

See how to use in *Defining and overriding settings*.

`index_params`

A dictionary of Sphinx options to override Sphinx settings of `index` for this particular index.

See how to use in *Defining and overriding settings*.

Field

Django-SphinxQL uses fields to identify which attributes from a Django model are indexed:

class `fields.`**`Field`**

A field to be added to an *`Index`*. A field is always mapped to a Django queryset, set on its initialization:

```
my_indexed_text = FieldType('text')  # Index.Meta.model contains `text =
...`
```

You can use both Django's F expressions or lookup expressions to index related fields or concatenate two fields. For instance, *TextField('article__text')*.

Fields are then mapped to a search field or an attribute of Sphinx:

- search fields are indexed for text search, and thus are used for textual searches with *`search()`*.
- attributes are used to filter and order the search results (see *`search_filter()`* and *`search_order_by()`*). They cannot be used in textual search.

The following fields are implemented in Django-SphinxQL:

- `Text`: a search field (Sphinx equivalent of no field declaration).
- `IndexedString`: attribute and search field (`sql_field_string`).
- `String`: (non-indexed) attribute for strings (`sql_attr_string`).
- `Date`: attribute for dates (`sql_attr_timestamp`).
- `DateTime`: attribute for datetimes (`sql_attr_timestamp`).
- `Float`: attribute for floats (`sql_attr_float`).
- `Bool`: attribute for booleans (`sql_attr_bool`).
- `Integer`: attribute for integers (`sql_attr_bigint`).

To simply index a Django field, use `Text`. If you need an attribute to filter or order your search results, use any of the attributes. Typically `IndexedString` is only needed if you want to use Sphinx without hitting Django's database (e.g. you redundantly store the data on Sphinx, query Sphinx and use the results of it.

Note that Sphinx `sql_attr_timestamp` is stored as a unix timestamp, so Django-SphinxQL only supports dates/times since 1970.

CHAPTER 2

Querying with Sphinx

This document presents the API of the Django-SphinxQL queryset, the high-level interface for interacting with Sphinx from Django.

SearchQuerySet

class sphinxql.query.**SearchQuerySet**

SearchQuerySet is a subclass of Django QuerySet to allow text-based search with Sphinx; This search is constructed by search* methods and is lazily applied to the Django QuerySet *before* it hits Django's database.

Formally, a SearchQuerySet is initialized with one parameter, the index it is bound to:

```
>>> q = SearchQuerySet(index, query=None, using=None)
```

that initializes Django's queryset from the *Index.Meta.model*.

The API of SearchQuerySet is the same as QuerySet, with the following additional methods:

- *search()*: for text searching
- *search_order_by()*: for ordering the results of the search
- *search_filter()*: for filtering the results of the search

If you don't use any of these methods, SearchQuerySet is equivalent to a Django QuerySet and can be directly replaced without any change.

When you apply *search()*, SearchQuerySet assumes you want to use Sphinx on it:

search_mode

Defaults to False. Defines whether Sphinx should be used by the *SearchQuerySet* prior to Django database hit. Automatically set to True when *search()* is used.

When *search_mode* is True, the queryset performs a search in Sphinx database with the query built from the search* methods before interacting with Django database:

- filtering done by *search()* and *search_filter()* are applied before Django's query, restricting the valid `id` in the Django's query.
- *search_order_by()* orders the results.

At most, `SearchQuerySet` does 1 database hit in Sphinx, followed by the Django hit. In *search_mode*, the `SearchQuerySet` has an upper limit:

max_search_count
A class attribute defining the maximum number of entries returned by the Sphinx hit. Currently hardcoded to 1000.

Notice that this implies that any search-based query has always at most count 1000 (can be less if Django filters some).

If Sphinx is used, model objects are annotated with an attribute `search_result` with the *Index* populated the values retrieved from Sphinx database.

Below, the full API is explained in detail:

search(**extended_queries*)
Adds a filter to text-search using Sphinx extended query syntax, defined by the strings `extended_queries`. Subsequent calls of this method concatenate the different `extended_query` with a space (equivalent to an `AND`).

This method automatically sets a search order according to relevance of the results given by the text search.

For instance:

```
>>> q = q.search('@text Hello world')
>>> q = q.filter(number__gt=2)
```

1. Searches for models with `Hello world` on the field `text`
2. orders them by most relevant first and retrieves the first *max_search_count* entries
3. filters the remaining entries with the Django query.

Notice that this method is orderless in the chain: Sphinx is always applied before the Django query.

search() supports arbitrary arguments to automatically restrict the search; the following are equivalent:

```
>>> q.search('@text Hello world @summary "my search"')
>>> q.search('@text Hello world', '@summary "my search"')
```

For convenience, here is a list of some operators (full list here):

- And: `' '` (a space)
- Or: `'|'` (`'hello | world'`)
- Not: `'-'` or `'!'` (e.g. `'hello -world'`)
- Mandatory first term, optional second term: `'MAYBE'` (e.g. `'hello MAYBE world'`)
- Phrase match: `'"<...>"'` (e.g. `'"hello world"'`)
- Before match: `'<<'` (e.g. `'hello << world'`)

search_order_by(**expressions*)
Adds `ORDER BY` clauses to Sphinx query. For example:

```
>>> q = q.search(a).search_order_by('-number')
```

will order first by the search relevance (*search()* added it) and then by `number` in decreasing order. Use `search_order_by()` to clear the ordering (default order is by `id`).

There are two built-in columns, `'@id'` and `'@relevance'`, that are used to order by Django `id` and by relevance of the results, respectively.

Notice that search ordering is applied *before* Django's query is performed. Yet, the final result (after Django query) is ordered according to Django ordering unless you didn't set any ordering to Django's query. For example:

```
>>> q = q.order_by('id').search(a)
```

orders the final results by `id` and:

```
>>> q = q.order_by('id').search(a).order_by()
```

orders the results by search relevance (because `order_by()` cleared Django's ordering).

In other words, the results are ordered by search ordering unless there is an explicit call of `order_by`.

search_filter(**conditions*, ***lookups*)
Adds a filter to the search query, allowing you to restrict the search results of the search.

`lookups` are like Django lookups for `filter`. Just remember that the field name must be defined on the *sphinxql.indexes.Index*.

`conditions` should be *Django-SphinxQL expressions* that return a boolean value (e.g. `>=`) and are used to produce more complex filters.

You can use `lookups` and `conditions` at the same time:

```
>>> q = q.search_filter(number__in=(2,3), C('number1')**2 > 10)
```

The method joins all and each `lookup` and `condition` with `AND`.

Like in Django, `"id__"` is reserved to indicate the object id (Sphinx shares the same ids as Django).

QuerySet

class query.**QuerySet**
`QuerySet` is a Django-equivalent `QuerySet` to indexes. I.e. contrary to *SearchQuerySet*, this QuerySet only interacts with the Sphinx database and returns instances of the Index. This can be useful when you need to present results of a search that don't need any extra data from Django.

The interface of this QuerySet is equivalent to Django QuerySet: it is lazy and allows chaining. However, the current implemented methods are limited:

search(**extended_queries*)
Same as *SearchQuerySet.search()*.

filter(**conditions*, ***lookups*)
Same as *SearchQuerySet.search_filter()*.

order_by(**expressions*)
Same as *SearchQuerySet.search_order_by()*.

count()
Same as Django's count.

CHAPTER 3

SphinxQL Queries

This section of the documentation explains how to construct expressions. To use queries with Django, see *Querying with Sphinx*.

Sphinx uses a dialect of SQL, SphinxQL, to perform operations on its database. Django-SphinxQL has a notation equivalent to Django to construct such expressions.

The basic unit of SphinxQL is a column. In Django-SphinxQL, a `Field` is a `Column` and thus the most explicit way to identify a column is to use:

```
>>> from myapp.indexes import PostIndex
>>> PostIndex.number  # a column
```

In a `query.SearchQuerySet`, you can use more implicit but simpler notations:

```
>>> PostIndex.objects.search_filter(number=2)
>>> from sphinxql.sql import C
>>> PostIndex.objects.search_filter(C('number') == 2)
>>> PostIndex.objects.search_filter(PostIndex.number == 2)
```

The first expression uses Django-equivalent lookups. The second uses `C('number')`, that is equivalent to Django F-expression and is resolved by the `SearchQuerySet` to `PostIndex.number` (or returns an error if `PostIndex` doesn't have a `Field` `number`).

Given a column, you can apply any Python operator (except bitwise) to it:

```
>>> my_expression = C('number')**2 + C('series')
>>> PostIndex.objects.search_filter(my_expression > 2)
>>> my_expression += 2
>>> my_expression = my_expression > 2  # it is now a condition
```

Warning: Django-SphinxQL still does not type-check operations: it can query `'hi hi' + 2 < 4` if you write a wrongly-typed expression.

To use SQL operators not defined in Python, you have two options:

```
>>> PostIndex.objects.search_filter(number__in=(2, 3))
>>> from sphinxql.sql import In
>>> PostIndex.objects.search_filter(C('number') |In| (2, 3))
```

Again, the first is the Django way and more implicit; the second is more explicit and lengthier, but allows you to create complex expressions, and uses Infix idiom.

The following operators are defined:

- `|And|` (separate conditions in `search_filter`)
- `|In|` (`__in`, like Django)
- `|NotIn|` (`__nin`)
- `|Between|` (`__range`, like Django)
- `|NotBetween|` (`__nrange`)

API references

Warning: This part of the documentation is still internal and subject to change/disappear.

SQLExpression

class `core.base.`**`SQLExpression`**

`SQLExpression` is the abstraction to build arbitrary SQL expressions. Almost everything in Django-SphinxQL is based on it: *`fields.Field`*, And, *`types.Value`*, etc.

It has most Python operators overridden such that an expression `C('foo') + 2` is converted into `Plus(C('foo'), Value(2))`, which can then be represented in SQL.

Values

class `types.`**`Value`**

Subclass of *`SQLExpression`* for constant values. Implemented by the following subclasses:

- `Bool`
- `Integer`
- `Float`
- `String`
- `Date`
- `DateTime`

Any `SQLExpression` that encounters a non-SQLExpression type tries to convert it to any of these types or raises a `TypeError`. For instance:

```
C('votes') < 10

is translated to ``SmallerThan(C('votes'), Integer(10))``.
```

`String` is always SQL-escaped.

Operations

class `sql.`**`BinaryOperation`**

Subclass of *`SQLExpression`* for binary operations. Implemented by the following subclasses:

- `Plus`
- `Subtract`
- `Multiply`
- `Divide`
- `Equal`
- `NotEqual`
- `And`
- `GreaterThan`
- `GreaterEqualThan`
- `LessThan`
- `LessEqualThan`

Other functions

- `In`, `NotIn`
- `Between`, `NotBetween`
- `Not`

Sphinx extended query syntax

class `sql.`**`Match`**

To filter results based on text, Sphinx defines a SQL keyword `MATCH()`. Inside this function, you can use its dedicated syntax to filter text against the Sphinx index. In Django-SphinxQL such filter is defined as a string inside a `Match` is a string:

```
>>> expression = Match('hello & world')
```

Since Sphinx only allows one `MATCH` per query, the public interface for using it is *`query.SearchQuerySet.search()`*, that automatically guarantees this.

CHAPTER 4

Sphinx Configuration

Configure Sphinx

Note: This part of the documentation requires a minimal understanding of Sphinx.

Running Sphinx requires a configuration file `sphinx.conf`, normally written by the user, that contains an arbitrary number of `sources` and `indexes`, one `indexer` and one `searchd`.

Django-SphinxQL provides an API to construct the `sphinx.conf` in Django: once you run Django with an *`Index`*, it automatically generates the `sphinx.conf` from your code, like Django builds a database when you run `migrate`.

Note: The `sphinx.conf` is modified by Django-SphinxQL from your code. It doesn't need to be added to the version control system.

Equivalently to Django, the `sources` and `indexes` of `sphinx.conf` are configured by an ORM (see *Indexes API*); `indexer` and `searchd` are configured by settings in Django settings.

Django-SphinxQL requires the user to define two settings:

- `INDEXES['path']`: the path of the database (a directory)
- `INDEXES['sphinx_path']`: the path for sphinx-related files (a directory)

For example:

```
INDEXES = {
    'path': os.path.join(BASE_DIR, '_index'),
    'sphinx_path': BASE_DIR,
}
```

Note: a) The paths must exist; b) `'path'` will contain the Sphinx database. c) ‘sphinx_path’ will contain 3 files:

pid file, searchd.log, and sphinx.conf.

Like Django, Django-SphinxQL already provides a (conservative) default configuration for Sphinx (e.g. it automatically sets Sphinx to assume unicode).

Default settings

Django-SphinxQL uses the following default settings:

```
'index_params': {
    'type': 'plain',
    'charset_type': 'utf-8'
}
'searchd_params': {
    'listen': '9306:mysql41',
    'pid_file': os.path.join(INDEXES['sphinx_path'], 'searchd.pid')
}
```

Defining and overriding settings

Django-SphinxQL applies settings in cascade, overriding previous settings if necessary, in the following order:

1. first, it uses Django-SphinxQL's default settings
2. them, it applies project-wise settings in `settings.INDEXES`, possibly overriding settings defined in 1.
3. finally, it applies the settings defined in the *Index.Meta*, possibly overriding settings in 2.

The project-wise settings use:

- `settings.INDEXES['searchd_params']`
- `settings.INDEXES['indexer_params']`
- `settings.INDEXES['index_params']`
- `settings.INDEXES['source_params']`

Each of them must be a dictionary that maps a Sphinx option (a Python string, e.g. `'charset_table'`) to a string or a tuple, depending whether the Sphinx option is single-valued or multi-valued.

For example:

```
INDEXES = {
        ...
        # sets U+00E0 to be considered part of the alphabet (and not be
        # considered a word separator) on all registered indexes.
        'index_params': {
            'charset_table': '0..9, A..Z->a..z, _, a..z, U+00E0'
        }
        # additionally to default, turns off query cache (see Sphinx docs)
        'source_params': {
            'sql_query_pre': ('SET SESSION query_cache_type=OFF',)
        }
        ...
}
```

You can also override settings of `source` and `index` only for a particular index by defining *`source_params`* and *`index_params`*.

Note: The options must be valid Sphinx options as defined in Sphinx documentation. Django-SphinxQL warns you if some option is not correct or is not valid.

`'index_params'` and `'source_params'` are used on every index configured; `'indexer_params'` and `'searchd_params'` are used in the `indexer` and `searchd` of `sphinx.conf`.

Configuration references (internal)

Warning: This part of the documentation is for internal use and subject to change.

Index configurator

class `configurators.`**`IndexConfigurator`**

This class is declared only once in Django-Sphinxql, and is responsible for mapping your *`indexes`* into a sphinx `sphinx.conf`.

This class has one entry point, *`register()`*, called automatically when *`Index`* is defined.

`register`(*index*)

Registers an *`Index`* in the configuration.

This is the entry point of this class to configure a new `Index`. A declaration of an `Index` automatically calls this method to register itself.

This method builds the source configuration and index configuration for the `index` and outputs the updated `sphinx.conf` to `INDEXES['sphinx_path']`.

On registering an index, *`register()`* gathers settings from three places:

- Django `settings`;
- *`Field`* of the index;
- *`Meta`* of the index.

From `django.settings`:

- `INDEXES['path']`: the directory where the database is created.
- `INDEXES['sphinx_path']`: the directory for sphinx-related files.
- `INDEXES['indexer_params']`: a dictionary with additional parameters for the *`IndexerConfiguration`*.
- `INDEXES['searchd_params']`: a dictionary with additional parameters for the *`SearchdConfiguration`*.
- `INDEXES['source_params']`: a dictionary with additional parameters for the *`SourceConfiguration`*.
- `INDEXES['index_params']`: a dictionary with additional parameters for the *`IndexConfiguration`*.

The set of available parameters can be found in Sphinx documentation.

From each field, the configurator uses its `type` and `model_attr`; from the *`Meta`*, it uses:

- *`model`*: the Django model the index is respective to.
- *`query`*: the Django query the index is populated from.
- *`range_step`*: the step for ranged queries (see Sphinx docs)

source configuration

class `sphinxql.configuration.configurations.`**`SourceConfiguration`**`(`*params*`)`

A class that can represent itself as an `source` of `sphinx.conf`.

It contains the parameters of the `source`. The argument of this class must be a dictionary mapping parameters to values.

This class always validates the parameters, raising an `ImproperlyConfigured` if any is wrong. The valid parameters are documented in Sphinx source configuration options.

It has one public method:

`format_output``()`

Returns a string with the parameters formatted according to the `sphinx.conf` syntax.

index configuration

class `sphinxql.configuration.configurations.`**`IndexConfiguration`**`(`*params*`)`

A class that can represent itself as an `index` of `sphinx.conf`.

It contains the parameters of the `index`. The argument of this class must be a dictionary mapping parameters to values.

This class always validates the parameters, raising an `ImproperlyConfigured` if any is wrong. The valid parameters are documented in Sphinx index configuration options.

It has one public method:

`format_output``()`

Returns a string with the parameters formatted according to the `sphinx.conf` syntax.

indexer configuration

class `sphinxql.configuration.configurations.`**`IndexerConfiguration`**`(`*params*`)`

A class that can represent itself as an `indexer` of `sphinx.conf`.

It contains the parameters of the `indexer`. The argument of this class must be a dictionary mapping parameters to values.

This class always validates the parameters, raising an `ImproperlyConfigured` if any is wrong. The valid parameters are documented in Sphinx indexer configuration options.

It has one public method:

`format_output``()`

Returns a string with the parameters formatted according to the `sphinx.conf` syntax.

searchd configuration

class `sphinxql.configuration.configurations.`**`SearchdConfiguration`**`(`*params*`)`

A class that can represent itself as the `searchd` of `sphinx.conf`.

It contains the parameters of the `searchd`. The argument of this class must be a dictionary mapping parameters to values.

This class always validates the parameters, raising an `ImproperlyConfigured` if any is wrong. The valid parameters are documented in Sphinx searchd configuration options.

It has one public method:

`format_output``()`

Returns a string with the parameters formatted according to the `sphinx.conf` syntax.

Index