nonrel-search - Full-text search for App Engine & NoSQL on Django

3 Like 4

Nonrel-search is a very simple Django-nonrel search engine for non-relational DBs based on `ListFields`. We have created a little sample project called Nonrel-search-testapp to demonstrate how to use nonrel-search.

Documentation Reference Source Download Tasks

Documentation

Key Features

Separate and multiple independent indexes on specific fields
You can specify which fields to index and create multiple independent indexes (each with its own fields) for one single model in separate modules.

Porter stemmers and word prefix search
Porter stemmers normalizes your words such that, for example, it doesn't matter whether you search a word in plural or singular. We include stemmers for English and German. Word prefix search allows for searching your data using an arbitrary beginning of a word (e.g. 'ho' matches 'house).

jQuery Auto-completion
Our auto-complete feature provides direct feedback about its functionality. We've taken care of handling all the little details that improve usability: For example, the user move his mouse around while holding down the mouse button, so he can make a correction if he mistakenly clicked on the wrong element.

Installation

Download nonrel-search or clone it via '`hg clone http://bitbucket.org/twanschik/nonrel-search`' and put the contained folder 'search ' into your project. Add 'search' to your `INSTALLED_APPS` in settings.py. Nonrel-search uses ListField such that it is dependend on djangotoolbox. Clone djangotoolbox via '`hg clone http://bitbucket.org/wkornewald/djangotoolbox`' and put its contained 'djangotoolbox' folder into your project. Add 'djangotoolbox' to your `INSTALLED_APPS` in settings.py. The same procedure has to be done for django-autoload since nonrel-search makes use of it. Note that you have to add Javascript / CSS files to your templates if you want use the auto-completion feature.

Indexing and searching your data

One way to make your entities searchable is to index them. The documentation will show you how to create such a search index for your model and how to use this index in order to search your entities. Throughout the documentation we'll use a model called Post to demonstrate the use of nonrel-search. So let's get started.

A post consists of a title, a content, an optional author, a category, and a post rating:

```
# post.models

from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=500)
    content = models.TextField()
    author = models.CharField(max_length=500)
    category = models.CharField(max_length=500)
    rating = models.IntegerField(choices=[(0, 'zero'), (1, 'one'), (2, 'two'),
        (3, 'three'), (4, 'four'), (5, 'five'), ], default=0)
```

As you can see, there is nothing special here. Let's say you want to make posts searchable only by

Follow @wkornewald, @twanschik and @johdoerr
Twitter conversations

zemanel @wkornewald hi, wanted to poke you about something related to nonrel but posted to the usergroup
6 days ago · reply · retweet · favorite

Join the conversation

Follow @wkornewald, @twanschik and @johdoerr

title, content and category leaving the author aside. This can be done by adding an index definition in a separate file called "post/search_indexes.py":

```
# post.search_indexes
import search
from search.core import porter_stemmer
from post.models import Post

# index used to retrieve posts using the title, content or the
# category.
search.register(Post, ('title', 'content','category', ),
    indexer=porter_stemmer)
```

Use the `register` function to add a search index for your posts. The first argument specifies the model to search for, the second argument specifies which fields to index. All remaining arguments are optional. The `indexer` argument specifies that we want to use the Porter Stemmer algorithm which normalizes all indexed words such that, for example, it doesn't matter whether you search a word in plural or singular. You can add multiple indexes for one model. To do so just register a new index and pass in the name for that index via the keyword argument 'search_index'.

You have to make sure that `search.register` will be executed. Therefor, you have to install django-autoload. nonrel-search provides a function called `autodiscover` which automatically searches the INSTALLED_APPS for "search_indexes.py" modules and registers all search indexes. `autodiscover` should be called from within your `settings.AUTOLOAD_SITECONF` module.

```
# AUTOLOAD_SITECONF module

# search for "search_indexes.py" in all installed apps
import search
search.autodiscover()
```

For more information on auto-loading see django-autoload.

Let's assume we have a post with the title 'Greetings' and the content 'Hello little world' saved in our database, then we can find it by calling

```
from search.core import search
results = search(Post, 'hello greeting')
# results[0] should be our first "Hello little world" post
```

nonrel-search automatically uses the search index 'search_index'. Here, "greeting" (without "s") matched the title because of the porter stemmer and obviously "hello" matched the content. The result is a Query like object you can add additional filters to:

```
# ...
results.filter(author='Itachi')
# now results should only contain entities with author set to 'Itachi'
```

Another possible indexer is 'startswith' which allows you to use word prefix search.

```
# post.search_indexes
import search
from search.core import startswith
from post.models import Post

# index used to retrieve posts via startswith
search.register(Post, ('title', 'content','category', ),
    indexer=startswith, search_index='autocomplete_index')
```

```
# post.views.py
...
results = search(Post, 'he', search_index='autocomplete_index')
```

Here we pass in the optional argument 'search_index' to tell nonrel-search to use the 'autocomplete_index' index for searching. Searching for 'hel' obviously matched the content because 'Hello' starts with 'hel'.

You should use word prefix search carefully because of some limitations on many nonrelational databses. Basically long text cannot be indexed this way.

Relation index and index backends

By default nonrel-search uses a relation index for storing indexed words in order to optimize later used gets from the database by only getting entities searched for, leaving the index aside. This is done by storing indexed words in a seperate entity. You can disable this behavior by setting 'relation_index' to 'False' when registering your search index:

```
# post.search_indexes
import search
from search.core import porter_stemmer
from post.models import Post

# index used to retrieve posts using the title, content or the
# category.
search.register(Post, ('title', 'content', 'category'),
    indexer=porter_stemmer, relation_index=False)
```

Disabling the relation_index results in storing the index in saved entities itself.

When using the relation index you can specify which fields to integrate into the relation index giving the possibility to make complexer queries. By default nonrel-search integrates all fields but you should optimize this otherwise you will store unnecessary data!

Again when using the relation index nonrel-search provides the possibility to use several behaviors in order to create / update indexes. By default nonrel-search creates / updates the index directly when saving entities. But let's say you want to create / update this index using a background task. This can be done by implementing a method '`update_relation_index`'. Nonrel-search will search your settings for a property called '`SEARCH_BACKEND`' which should hold a string containing the module of where your custom method is implemented. Here is an example of how to set a custom backend:

```
# settings.py
...
SEARCH_BACKEND = 'search.backends.gae_background_tasks'
...
```

Nonrel-search provides two build-in backends. `search.backends.immediate_update` updates the index directly when saving entities and can be used for any database e.g. MongoDB, App Engine, ... `search.backends.gae_background_tasks` is an backend which uses App Engine's background tasks to update the index. Therefor it only can be used with App Engine set as your database.

If you want to see a more complete example of how to use nonrel-search take a look at the demo application. There you can find all remaining things related to a complete Django app e.g. the urls, templates and views. Don't forget to copy / clone nonrel-search, djangotoolbox, djangoappengine and django-nonrel into the demo application. See Native Django on App Engine for installation instructions for djangoappengine and django-nonrel.

Now you should have an understanding of how to use nonrel-search. For more information see the reference.

Reference

```
from search import register
register(model, fields_to_index, indexer, splitter, relation_index,
    integrate, filters, language, search_index)
```

Take a look at the documentation to see an example.

This is the heart of the search engine. The `register` function takes care of creating / adding an index to your Model that is to create / add a subclass of `ListField`.

Arguments

model: Model class (required)
This is the model for which the index should be created.

fields_to_index: string or tuple of strings (required)
The name of the field that should be indexed or a tuple of field names.

indexer: function (default: None)
A function taking a list of words and returning the list of words that should be indexed. See below for the list of indexers. If no indexer is provided the words are indexed as-is.

splitter: function (default: `default_splitter`)
Normally, you won't need the `splitter` argument because there is just one splitter. The splitter takes the whole indexable text as a string and is responsible for splitting it into words. The default splitter tries to enhance the search experience: if we have the word "mega-man" it'll be splitted as

"mega", "man", and "megaman", so users can find the resulting word even if they enter without the "-".

relation_index: boolean (default: True)
Specifies whether to store the index in a separate entity. This saves a lot of resources when running the query because the underlying `ListField` doesn't have to be transferred and deserialized. If you want to filter results by additional fields you have to specify their names in the `integrate` parameter. By default, all fields are integrated, so you are prepared for unexpected cases, but normally you should optimize database usage by limiting the integrated fields.

integrate: string or tuple of strings (default: integrates all fields)
This can only be used together with `relation_index=True`. It allows for creating an index that can be filtered by the integrated fields.

filers: dictionary of django database filter rules (optional)
You can specify database filters like `{'is_active__exact':True, 'is_banned': False}` in order to control whether an index should be generated, at all when saving entities. This is useful if you want to reduce the amount of indexed data.

language: string or function (default: `site_language` handler)
In order to tell the indexer (e.g., `porter_stemmer`) which language your current entity's content has you can either provide a string with the language code (e.g., "en") or you can provide a function which tries to detect the language for you. The default `site_language` handler tries to access the following entity fields in order to determine the content language: "language", "lang". If nothing works it falls back to your `settings.LANGUAGE_CODE`.

search_index: string (default: 'search_index')
This is the name of the index created by register. Later this name can be selected to be used for searching.

Built-in Indexers

The following indexers are integrated into the `search.core` module:

porter_stemmer An indexer which uses the Porter Stemmer algorithm. It allows for searching words independent of whether they're written in plural or singular. We currently provide algorithms for English and German.

porter_stemmer_non_stop This is like the porter_stemmer indexer, but it removes stop words from the content ("a", "and", "or").

startswith This allows for word prefix search (it basically simulates an SQL "LIKE ...%" filter). Don't use this for long texts, though, because each word is stored as multiple substrings ("house" becomes "h", "ho", "hou", "hous", "house").

Searching

```
from search.core import search
results = search(model, query, search_index, language)
```

model: Model class (required)
The model to search for.

query: string (required)
The name text that you want to search for (e.g. "cheap laptops").

search_index: string (optional, default: 'search_index)
The index to use for searching

language: string (default: `settings.LANGUAGE_CODE`)
In order to tell the indexer (e.g., `porter_stemmer`) which language your current entity's content has you have to provide a string with the language code (e.g., "en").

Index Backends

You can use several backends to specify when to index your data when using the relation index. This has to be set in settings.py

```
settings.SEARCH_BACKEND = 'search.backends.immediate_update'
```

Available backends are:

search.backends.immediate_update (default)
Updates the index directly when saving entities. Can be used with any nonrelational database.

search.backends.gae_background_tasks
Updates the index in a App Engine background task. Can only be used in combination with App Engine set as the database used for Django.

Custom backends
If you want to use a custom backend you have to implement a method called

'update_relation_index' getting the following arguments:

search_index_field
The index which has to be updated. At some point in your custom method you should call search_index_field.update_relation_index.

parent_pk
The primary key of the entity which has changed.

delete
delete indicates wether the parent has been deleted or not.

LiveSearchField

```
from search.forms import LiveSearchField
LiveSearchField(src, multiple_values, select_first, auto_fill,
    must_match, match_contains)
```

This form field can be used to easily add live-search to your site.

Arguments
Note: you can provide additional keyword arguments which will be passed directly to the underlying CharField.

src: string (required)
This is the URI of the live-search view.

multiple_values: boolean (default: False)
Specifies whether the user should be able to enter multiple comma-separated values into a single input field (e.g., multiple tags).

select_first: boolean (default: False)
Automatically preselects the first result.

auto_fill: boolean (default: False)
Fills the input field with the first result's value while typing. Note: By setting this to True, match_contains will be ignored and set to False.

must_match: boolean (default: False)
Disallows to enter values that are not part of the results.

match_contains: boolean (default: True)
Specifies whether to match any part of the string ("contains") or only from the beginning of the string ("startswith"). Note: When setting auto_fill=True, match_contains will be ignored and set to False.

live_search_results

```
from search.views import live_search_results
live_search_results(request, model, index, filters, chain_sort, limit,
        result_item_formatting, query_converter, converter, redirect)
```

This view returns its search results as a JSON response (a list of dicts) for use with our auto-complete plug-in.

It will automatically use the language from request.LANGUAGE_CODE and pass that to the SearchIndexProperty's search() function.

The JSON response has the following format:

```
[ // List of people
  { // Result 1
    'result': 'Joe Hacker<injected-tag>', // String that should be copied into
                                // input when field item is selected (not escaped)
    'value': 'Joe Hacker&lt;injected-tag&gt;', // String as displayed in drop-down
                                // results (should be escaped)
    'data':  {'link': '/user/joe-hacker/'}  // Optional; specifies where user should
                                // be redirected when selecting a result
  },
  { // Result 2
    'result': 'Joe User',
    'value': 'Joe User',
    'data':  {}  // When clicking this result you don't get redirected. Instead, the
           // 'result' is copied into the input field.
  },
  ...
]
```

Arguments
model: Model class (required)
The model class you want to search on.

index: string (optional, default='search_index')
The name of the SearchIndexProperty you want to use.

filters: tuple of datastore filter rules (optional)
Allows for adding datastore filters to the query. For example: *filters=('is_active =', True, 'is_banned =', False)*

limit: integer (default: 30; jQuery auto-completer default: 10)
Maximum number of results to return. This can be overridden via a GET parameter named "limit", but only if that limit is smaller or equal. Never set this value lower than what the jQuery auto-completer (default: 10) expects because then you get invalid result handling.

result_item_formatting: function (optional)
Gets a result entity and returns the corresponding dictionary item which represents that entity in the JSON response. If you don't specify any function, value will be the escaped value of the first indexed property and result will be the unescaped value of the same property.

query_converter: function (optional)New
Gets the request and the query object matching the results and returns a new query object. This allows for modifying the query (e.g., needed for geomodel).

converter: function (optional)
Gets the whole list of results and returns a new list of results. This can be used for optimizations (e.g., fetch all reference properties at once before they are accessed).

redirect: boolean (default: False)
Specifies whether the user should be redirected to the result item's get_absolute_url() when selecting a result or whether the item's value should merely be copied into the input field.

Differences to gae-search

Filter

Nonrel-search uses Django style filters instead of App Engine style filters that is you have to pass in a dictionary of filter rules instead of a list: Here is an example:

```
# in gae-search
filters=('title =', 'hello', 'author =', 'Itachi')
# in nonrel-search
filters={'title':'hello', 'author': 'Itachi'}
```

Additionally you do not pass in filters to search function calls anymore. Instead you have to chain these filters:

```
# in gae-search
search(Post, 'hello', filters=('author =', 'Itachi'))
# in nonrel-search
search(Post, 'hello').filter(author='Itachi')
```

No values index

Gae-search doesn't supports the values index. This may come as an independent backend layer for django-nonrel.

language support

Nonrel-search does not check if a given entity has a language-specific site. The language can only be set via a field called "language", "lang" or `settings.LANGUAGE_CODE`.