
Haystack Documentation

Release 2.0.0

Daniel Lindsley

August 28, 2016

1	Getting Started	3
1.1	Getting Started with Haystack	3
1.2	Views & Forms	9
1.3	Template Tags	13
1.4	Glossary	14
1.5	Management Commands	15
1.6	(In)Frequently Asked Questions	18
1.7	Sites Using Haystack	19
1.8	Haystack-Related Applications	24
1.9	Installing Search Engines	26
1.10	Debugging Haystack	29
1.11	Migrating From Haystack 1.X to Haystack 2.X	31
1.12	Contributing	35
2	Advanced Uses	39
2.1	Best Practices	39
2.2	Highlighting	43
2.3	Faceting	44
2.4	Autocomplete	48
2.5	Boost	51
2.6	Signal Processors	53
2.7	Multiple Indexes	54
2.8	Rich Content Extraction	57
2.9	Spatial Search	58
3	Reference	65
3.1	SearchQuerySet API	65
3.2	SearchIndex API	79
3.3	Input Types	88
3.4	SearchField API	90
3.5	SearchResult API	95
3.6	SearchQuery API	96
3.7	SearchBackend API	101
3.8	Architecture Overview	102
3.9	Backend Support	104
3.10	Haystack Settings	106
3.11	Utilities	110

4	Developing	111
4.1	Running Tests	111
4.2	Creating New Backends	112
5	Requirements	115
6	Commercial Support	117

Haystack provides modular search for Django. It features a unified, familiar API that allows you to plug in different search backends (such as [Solr](#), [Elasticsearch](#), [Whoosh](#), [Xapian](#), etc.) without having to modify your code.

Note: This documentation represents the development version of Haystack. For old versions of the documentation: [1.2](#), [1.1](#).

Getting Started

If you're new to Haystack, you may want to start with these documents to get you up and running:

1.1 Getting Started with Haystack

Search is a topic of ever increasing importance. Users increasingly rely on search to separate signal from noise and find what they're looking for quickly. In addition, search can provide insight into what things are popular (many searches), what things are difficult to find on the site and ways you can improve the site better.

To this end, Haystack tries to make integrating custom search as easy as possible while being flexible/powerful enough to handle more advanced use cases.

Haystack is a reusable app (that is, it relies only on its own code and focuses on providing just search) that plays nicely with both apps you control as well as third-party apps (such as `django.contrib.*`) without having to modify the sources.

Haystack also does pluggable backends (much like Django's database layer), so virtually all of the code you write ought to be portable between whichever search engine you choose.

Note: If you hit a stumbling block, there is both a [mailing list](#) and [#haystack](#) on [irc.freenode.net](#) to get help.

Note: You can participate in and/or track the development of Haystack by subscribing to the [development mailing list](#).

This tutorial assumes that you have a basic familiarity with the various major parts of Django (models/forms/views/settings/URLconfs) and tailored to the typical use case. There are shortcuts available as well as hooks for much more advanced setups, but those will not be covered here.

For example purposes, we'll be adding search functionality to a simple note-taking application. Here is `myapp/models.py`:

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

class Note(models.Model):
    user = models.ForeignKey(User)
    pub_date = models.DateTimeField()
    title = models.CharField(max_length=200)
```

```
body = models.TextField()

def __unicode__(self):
    return self.title
```

Finally, before starting with Haystack, you will want to choose a search backend to get started. There is a quick-start guide to [Installing Search Engines](#), though you may want to defer to each engine's official instructions.

1.1.1 Configuration

Add Haystack To `INSTALLED_APPS`

As with most Django applications, you should add Haystack to the `INSTALLED_APPS` within your settings file (usually `settings.py`).

Example:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',

    # Added.
    'haystack',

    # Then your usual apps...
    'blog',
]
```

Modify Your `settings.py`

Within your `settings.py`, you'll need to add a setting to indicate where your site configuration file will live and which backend to use, as well as other settings for that backend.

`HAYSTACK_CONNECTIONS` is a required setting and should be at least one of the following:

Solr

Example:

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.solr_backend.SolrEngine',
        'URL': 'http://127.0.0.1:8983/solr'
        # ...or for multicore...
        # 'URL': 'http://127.0.0.1:8983/solr/mysite',
    },
}
```

Elasticsearch

Example:


```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.elasticsearch_backend.ElasticsearchSearchEngine',
        'URL': 'http://127.0.0.1:9200/',
        'INDEX_NAME': 'haystack',
    },
}
```

Whoosh

Requires setting `PATH` to the place on your filesystem where the Whoosh index should be located. Standard warnings about permissions and keeping it out of a place your webserver may serve documents out of apply.

Example:

```
import os
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.whoosh_backend.WhooshEngine',
        'PATH': os.path.join(os.path.dirname(__file__), 'whoosh_index'),
    },
}
```

Xapian

First, install the Xapian backend (via <http://github.com/notanumber/xapian-haystack/tree/master>) per the instructions included with the backend.

Requires setting `PATH` to the place on your filesystem where the Xapian index should be located. Standard warnings about permissions and keeping it out of a place your webserver may serve documents out of apply.

Example:

```
import os
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.xapian_backend.XapianEngine',
        'PATH': os.path.join(os.path.dirname(__file__), 'xapian_index'),
    },
}
```

Simple

The simple backend using very basic matching via the database itself. It's not recommended for production use but it will return results.

Warning: This backend does *NOT* work like the other backends do. Data preparation does nothing & advanced filtering calls do not work. You really probably don't want this unless you're in an environment where you just want to silence Haystack.

Example:

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.simple_backend.SimpleEngine',
    },
}
```

1.1.2 Handling Data

Creating SearchIndexes

SearchIndex objects are the way Haystack determines what data should be placed in the search index and handles the flow of data in. You can think of them as being similar to Django Models or Forms in that they are field-based and manipulate/store data.

You generally create a unique SearchIndex for each type of Model you wish to index, though you can reuse the same SearchIndex between different models if you take care in doing so and your field names are very standardized.

To build a SearchIndex, all that's necessary is to subclass both `indexes.SearchIndex` & `indexes.Indexable`, define the fields you want to store data with and define a `get_model` method.

We'll create the following `NoteIndex` to correspond to our `Note` model. This code generally goes in a `search_indexes.py` file within the app it applies to, though that is not required. This allows Haystack to automatically pick it up. The `NoteIndex` should look like:

```
import datetime
from haystack import indexes
from myapp.models import Note

class NoteIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    author = indexes.CharField(model_attr='user')
    pub_date = indexes.DateTimeField(model_attr='pub_date')

    def get_model(self):
        return Note

    def index_queryset(self, using=None):
        """Used when the entire index for model is updated."""
        return self.get_model().objects.filter(pub_date__lte=datetime.datetime.now())
```

Every SearchIndex requires there be one (and only one) field with `document=True`. This indicates to both Haystack and the search engine about which field is the primary field for searching within.

Warning: When you choose a `document=True` field, it should be consistently named across all of your SearchIndex classes to avoid confusing the backend. The convention is to name this field `text`. There is nothing special about the `text` field name used in all of the examples. It could be anything; you could call it `pink_polka_dot` and it won't matter. It's simply a convention to call it `text`.

Additionally, we're providing `use_template=True` on the `text` field. This allows us to use a data template (rather than error prone concatenation) to build the document the search engine will use in searching. You'll need to create a new template inside your template directory called `search/indexes/myapp/note_text.txt` and place the following inside:

```

{{ object.title }}
{{ object.user.get_full_name }}
{{ object.body }}

```

In addition, we added several other fields (`author` and `pub_date`). These are useful when you want to provide additional filtering options. Haystack comes with a variety of `SearchField` classes to handle most types of data.

A common theme is to allow admin users to add future content but have it not display on the site until that future date is reached. We specify a custom `index_queryset` method to prevent those future items from being indexed.

1.1.3 Setting Up The Views

Add The SearchView To Your URLconf

Within your URLconf, add the following line:

```
(r'^search/', include('haystack.urls')),
```

This will pull in the default URLconf for Haystack. It consists of a single URLconf that points to a `SearchView` instance. You can change this class's behavior by passing it any of several keyword arguments or override it entirely with your own view.

Search Template

Your search template (`search/search.html` for the default case) will likely be very simple. The following is enough to get going (your template/block names will likely differ):

```

{% extends 'base.html' %}

{% block content %}
    <h2>Search</h2>

    <form method="get" action=".">
        <table>
            {{ form.as_table }}
            <tr>
                <td>&nbsp;</td>
                <td>
                    <input type="submit" value="Search">
                </td>
            </tr>
        </table>

        {% if query %}
            <h3>Results</h3>

            {% for result in page.object_list %}
                <p>
                    <a href="{{ result.object.get_absolute_url }}">{{ result.object.title }}</a>
                </p>
            {% empty %}
                <p>No results found.</p>
            {% endfor %}

            {% if page.has_previous or page.has_next %}
                <div>

```

```
        {% if page.has_previous %}<a href="?q={{ query }}&page={{ page.previous_page_number }}&page_size={{ page_size }}">Previous</a>
        |
        {% if page.has_next %}<a href="?q={{ query }}&page={{ page.next_page_number }}&page_size={{ page_size }}">Next</a>
    </div>
    {% endif %}
{% else %}
    {# Show some example queries to run, maybe query syntax, something else? #}
{% endif %}
</form>
{% endblock %}
```

Note that the `page.object_list` is actually a list of `SearchResult` objects instead of individual models. These objects have all the data returned from that record within the search index as well as score. They can also directly access the model for the result via `{{ result.object }}`. So the `{{ result.object.title }}` uses the actual `Note` object in the database and accesses its `title` field.

Reindex

The final step, now that you have everything setup, is to put your data in from your database into the search index. Haystack ships with a management command to make this process easy.

Note: If you're using the Solr backend, you have an extra step. Solr's configuration is XML-based, so you'll need to manually regenerate the schema. You should run `./manage.py build_solr_schema` first, drop the XML output in your Solr's `schema.xml` file and restart your Solr server.

Simply run `./manage.py rebuild_index`. You'll get some totals of how many models were processed and placed in the index.

Note: Using the standard `SearchIndex`, your search index content is only updated whenever you run either `./manage.py update_index` or start afresh with `./manage.py rebuild_index`.

You should cron up a `./manage.py update_index` job at whatever interval works best for your site (using `--age=<num_hours>` reduces the number of things to update).

Alternatively, if you have low traffic and/or your search engine can handle it, the `RealtimeSignalProcessor` automatically handles updates/deletes for you.

1.1.4 Complete!

You can now visit the search section of your site, enter a search query and receive search results back for the query! Congratulations!

1.1.5 What's Next?

This tutorial just scratches the surface of what Haystack provides. The `SearchQuerySet` is the underpinning of all search in Haystack and provides a powerful, `QuerySet`-like API (see [SearchQuerySet API](#)). You can use much more complicated `SearchForms/SearchViews` to give users a better UI (see [Views & Forms](#)). And the [Best Practices](#) provides insight into non-obvious or advanced usages of Haystack.

1.2 Views & Forms

Haystack comes with some default, simple views & forms to help you get started and to cover the common cases. Included is a way to provide:

- Basic, query-only search.
- Search by models.
- Search with basic highlighted results.
- Faceted search.
- Search by models with basic highlighted results.

Most processing is done by the forms provided by Haystack via the `search` method. As a result, all but the faceted types (see [Faceting](#)) use the standard `SearchView`.

There is very little coupling between the forms & the views (other than relying on the existence of a `search` method on the form), so you may interchangeably use forms and/or views anywhere within your own code.

1.2.1 Forms

SearchForm

The most basic of the form types, this form consists of a single field, the `q` field (for query). Upon searching, the form will take the cleaned contents of the `q` field and perform an `auto_query` on either the custom `SearchQuerySet` you provide or off a default `SearchQuerySet`.

To customize the `SearchQuerySet` the form will use, pass it a `searchqueryset` parameter to the constructor with the `SearchQuerySet` you'd like to use. If using this form in conjunction with a `SearchView`, the form will receive whatever `SearchQuerySet` you provide to the view with no additional work needed.

The `SearchForm` also accepts a `load_all` parameter (`True` or `False`), which determines how the database is queried when iterating through the results. This also is received automatically from the `SearchView`.

All other forms in Haystack inherit (either directly or indirectly) from this form.

HighlightedSearchForm

Identical to the `SearchForm` except that it tags the `highlight` method on to the end of the `SearchQuerySet` to enable highlighted results.

ModelSearchForm

This form adds new fields to form. It iterates through all registered models for the current `SearchSite` and provides a checkbox for each one. If no models are selected, all types will show up in the results.

HighlightedModelSearchForm

Identical to the `ModelSearchForm` except that it tags the `highlight` method on to the end of the `SearchQuerySet` to enable highlighted results on the selected models.

FacetedSearchForm

Identical to the `SearchForm` except that it adds a hidden `selected_facets` field onto the form, allowing the form to narrow the results based on the facets chosen by the user.

Creating Your Own Form

The simplest way to go about creating your own form is to inherit from `SearchForm` (or the desired parent) and extend the `search` method. By doing this, you save yourself most of the work of handling data correctly and stay API compatible with the `SearchView`.

For example, let's say you're providing search with a user-selectable date range associated with it. You might create a form that looked as follows:

```
from django import forms
from haystack.forms import SearchForm

class DateRangeSearchForm(SearchForm):
    start_date = forms.DateField(required=False)
    end_date = forms.DateField(required=False)

    def search(self):
        # First, store the SearchQuerySet received from other processing.
        sqs = super(DateRangeSearchForm, self).search()

        if not self.is_valid():
            return self.no_query_found()

        # Check to see if a start_date was chosen.
        if self.cleaned_data['start_date']:
            sqs = sqs.filter(pub_date__gte=self.cleaned_data['start_date'])

        # Check to see if an end_date was chosen.
        if self.cleaned_data['end_date']:
            sqs = sqs.filter(pub_date__lte=self.cleaned_data['end_date'])

        return sqs
```

This form adds two new fields for (optionally) choosing the start and end dates. Within the `search` method, we grab the results from the parent form's processing. Then, if a user has selected a start and/or end date, we apply that filtering. Finally, we simply return the `SearchQuerySet`.

1.2.2 Views

Haystack comes bundled with three views, the class-based views (`SearchView` & `FacetedSearchView`) and a traditional functional view (`basic_search`).

The class-based views provide for easy extension should you need to alter the way a view works. Except in the case of faceting (again, see [Faceting](#)), the `SearchView` works interchangeably with all other forms provided by Haystack.

The functional view provides an example of how Haystack can be used in more traditional settings or as an example of how to write a more complex custom view. It is also thread-safe.

```
SearchView(template=None, load_all=True, form_class=None, searchqueryset=None,
context_class=RequestContext, results_per_page=None)
```

The `SearchView` is designed to be easy/flexible enough to override common changes as well as being internally abstracted so that only altering a specific portion of the code should be easy to do.

Without touching any of the internals of the `SearchView`, you can modify which template is used, which form class should be instantiated to search with, what `SearchQuerySet` to use in the event you wish to pre-filter the results, what Context-style object to use in the response and the `load_all` performance optimization to reduce hits on the database. These options can (and generally should) be overridden at the URLconf level. For example, to have a custom search limited to the 'John' author, displaying all models to search by and specifying a custom template (`my/special/path/john_search.html`), your URLconf should look something like:

```
from django.conf.urls.defaults import *
from haystack.forms import ModelSearchForm
from haystack.query import SearchQuerySet
from haystack.views import SearchView

sqs = SearchQuerySet().filter(author='john')

# Without threading...
urlpatterns = patterns('haystack.views',
    url(r'^$', SearchView(
        template='my/special/path/john_search.html',
        searchqueryset=sqs,
        form_class=SearchForm
    ), name='haystack_search'),
)

# With threading...
from haystack.views import SearchView, search_view_factory

urlpatterns = patterns('haystack.views',
    url(r'^$', search_view_factory(
        view_class=SearchView,
        template='my/special/path/john_search.html',
        searchqueryset=sqs,
        form_class=ModelSearchForm
    ), name='haystack_search'),
)
```

Warning: The standard `SearchView` is not thread-safe. Use the `search_view_factory` function, which returns thread-safe instances of `SearchView`.

By default, if you don't specify a `form_class`, the view will use the `haystack.forms.ModelSearchForm` form.

Beyond this customizations, you can create your own `SearchView` and extend/override the following methods to change the functionality.

```
__call__(self, request)
```

Generates the actual response to the search.

Relies on internal, overridable methods to construct the response. You generally should avoid altering this method unless you need to change the flow of the methods or to add a new method into the processing.

```
build_form(self, form_kwargs=None)
```

Instantiates the form the class should use to process the search query.

Optionally accepts a dictionary of parameters that are passed on to the form's `__init__`. You can use this to lightly customize the form.

You should override this if you write a custom form that needs special parameters for instantiation.

```
get_query(self)
```

Returns the query provided by the user.

Returns an empty string if the query is invalid. This pulls the cleaned query from the form, via the `q` field, for use elsewhere within the `SearchView`. This is used to populate the `query` context variable.

```
get_results(self)
```

Fetches the results via the form.

Returns an empty list if there's no query to search with. This method relies on the form to do the heavy lifting as much as possible.

```
build_page(self)
```

Paginates the results appropriately.

In case someone does not want to use Django's built-in pagination, it should be a simple matter to override this method to do what they would like.

```
extra_context(self)
```

Allows the addition of more context variables as needed. Must return a dictionary whose contents will add to or overwrite the other variables in the context.

```
create_response(self)
```

Generates the actual `HttpResponse` to send back to the user. It builds the page, creates the context and renders the response for all the aforementioned processing.

```
basic_search(request, template='search/search.html', load_all=True,  
form_class=ModelSearchForm, searchqueryset=None, context_class=RequestContext,  
extra_context=None, results_per_page=None)
```

The `basic_search` tries to provide most of the same functionality as the class-based views but resembles a more traditional generic view. It's both a working view if you prefer not to use the class-based views as well as a good starting point for writing highly custom views.

Since it is all one function, the only means of extension are passing in `kwargs`, similar to the way generic views work.

Creating Your Own View

As with the forms, inheritance is likely your best bet. In this case, the `FacetedSearchView` is a perfect example of how to extend the existing `SearchView`. The complete code for the `FacetedSearchView` looks like:

```
class FacetedSearchView(SearchView):
    def extra_context(self):
        extra = super(FacetedSearchView, self).extra_context()

        if self.results == []:
            extra['facets'] = self.form.search().facet_counts()
        else:
            extra['facets'] = self.results.facet_counts()

        return extra
```

It updates the name of the class (generally for documentation purposes) and adds the facets from the `SearchQuerySet` to the context as the `facets` variable. As with the custom form example above, it relies on the parent class to handle most of the processing and extends that only where needed.

1.3 Template Tags

Haystack comes with a couple common template tags to make using some of its special features available to templates.

1.3.1 highlight

Takes a block of text and highlights words from a provided query within that block of text. Optionally accepts arguments to provide the HTML tag to wrap highlighted word in, a CSS class to use with the tag and a maximum length of the blurb in characters.

The defaults are `span` for the HTML tag, `highlighted` for the CSS class and 200 characters for the excerpt.

Syntax:

```
{% highlight <text_block> with <query> [css_class "class_name"] [html_tag "span"] [max_length 200] %}
```

Example:

```
# Highlight summary with default behavior.
{% highlight result.summary with request.query %}

# Highlight summary but wrap highlighted words with a div and the
# following CSS class.
{% highlight result.summary with request.query html_tag "div" class "highlight_me_please" %}

# Highlight summary but only show 40 words.
{% highlight result.summary with request.query max_length 40 %}
```

The highlighter used by this tag can be overridden as needed. See the [Highlighting](#) documentation for more information.

1.3.2 more_like_this

Fetches similar items from the search index to find content that is similar to the provided model's content.

Note: This requires a backend that has More Like This built-in.

Syntax:

```
{% more_like_this model_instance as varname [for app_label.model_name, app_label.model_name, ...] [limit]
```

Example:

```
# Pull a full SearchQuerySet (lazy loaded) of similar content.
{% more_like_this entry as related_content %}

# Pull just the top 5 similar pieces of content.
{% more_like_this entry as related_content limit 5 %}

# Pull just the top 5 similar entries or comments.
{% more_like_this entry as related_content for "blog.entry,comments.comment" limit 5 %}
```

This tag behaves exactly like `SearchQuerySet.more_like_this`, so all notes in that regard apply here as well.

1.4 Glossary

Search is a domain full of it's own jargon and definitions. As this may be an unfamiliar territory to many developers, what follows are some commonly used terms and what they mean.

Engine An engine, for the purposes of Haystack, is a third-party search solution. It might be a full service (i.e. [Solr](#)) or a library to build an engine with (i.e. [Whoosh](#))

Index The datastore used by the engine is called an index. Its structure can vary wildly between engines but commonly they resemble a document store. This is the source of all information in Haystack.

Document A document is essentially a record within the index. It usually contains at least one blob of text that serves as the primary content the engine searches and may have additional data hung off it.

Corpus A term for a collection of documents. When talking about the documents stored by the engine (rather than the technical implementation of the storage), this term is commonly used.

Field Within the index, each document may store extra data with the main content as a field. Also sometimes called an attribute, this usually represents metadata or extra content about the document. Haystack can use these fields for filtering and display.

Term A term is generally a single word (or word-like) string of characters used in a search query.

Stemming A means of determining if a word has any root words. This varies by language, but in English, this generally consists of removing plurals, an action form of the word, et cetera. For instance, in English, 'giraffes' would stem to 'giraffe'. Similarly, 'exclamation' would stem to 'exclaim'. This is useful for finding variants of the word that may appear in other documents.

Boost Boost provides a means to take a term or phrase from a search query and alter the relevance of a result based on if that term is found in the result, a form of weighting. For instance, if you wanted to more heavily weight results that included the word 'zebra', you'd specify a boost for that term within the query.

More Like This Incorporating techniques from information retrieval and artificial intelligence, More Like This is a technique for finding other documents within the index that closely resemble the document in question. This is useful for programmatically generating a list of similar content for a user to browse based on the current document they are viewing.

Faceting Faceting is a way to provide insight to the user into the contents of your corpus. In its simplest form, it is a set of document counts returned with results when performing a query. These counts can be used as feedback

for the user, allowing the user to choose interesting aspects of their search results and “drill down” into those results.

An example might be providing a facet on an `author` field, providing back a list of authors and the number of documents in the index they wrote. This could be presented to the user with a link, allowing the user to click and narrow their original search to all results by that author.

1.5 Management Commands

Haystack comes with several management commands to make working with Haystack easier.

1.5.1 `clear_index`

The `clear_index` command wipes out your entire search index. Use with caution. In addition to the standard management command options, it accepts the following arguments:

```
``--noinput``:
    If provided, the interactive prompts are skipped and the index is
    uncerimoniously wiped out.
``--verbosity``:
    Accepted but ignored.
``--using``:
    If provided, determines which connection should be used. Default is
    ``default``.
```

By default, this is an **INTERACTIVE** command and assumes that you do **NOT** wish to delete the entire index.

Warning: Depending on the backend you’re using, this may simply delete the entire directory, so be sure your `HAYSTACK_CONNECTIONS[<alias>][‘PATH’]` setting is correctly pointed at just the index directory.

1.5.2 `update_index`

Note: If you use the `--start/--end` flags on this command, you’ll need to install `dateutil` to handle the datetime parsing.

The `update_index` command will freshen all of the content in your index. It iterates through all indexed models and updates the records in the index. In addition to the standard management command options, it accepts the following arguments:

```
``--age``:
    Number of hours back to consider objects new. Useful for nightly
    reindexes (``--age=24``). Requires ``SearchIndexes`` to implement
    the ``get_updated_field`` method. Default is ``None``.
``--start``:
    The start date for indexing within. Can be any dateutil-parsable string,
    recommended to be YYYY-MM-DDTHH:MM:SS. Requires ``SearchIndexes`` to
    implement the ``get_updated_field`` method. Default is ``None``.
``--end``:
    The end date for indexing within. Can be any dateutil-parsable string,
    recommended to be YYYY-MM-DDTHH:MM:SS. Requires ``SearchIndexes`` to
    implement the ``get_updated_field`` method. Default is ``None``.
```

```
``--batch-size``:
    Number of items to index at once. Default is 1000.
``--remove``:
    Remove objects from the index that are no longer present in the
    database.
``--workers``:
    Allows for the use multiple workers to parallelize indexing. Requires
    ``multiprocessing``.
``--verbosity``:
    If provided, dumps out more information about what's being done.

    * ``0`` = No output
    * ``1`` = Minimal output describing what models were indexed
      and how many records.
    * ``2`` = Full output, including everything from ``1`` plus output
      on each batch that is indexed, which is useful when debugging.
``--using``:
    If provided, determines which connection should be used. Default is
    ``default``.
```

Examples:

```
# Update everything.
./manage.py update_index --settings=settings.prod

# Update everything with lots of information about what's going on.
./manage.py update_index --settings=settings.prod --verbosity=2

# Update everything, cleaning up after deleted models.
./manage.py update_index --remove --settings=settings.prod

# Update everything changed in the last 2 hours.
./manage.py update_index --age=2 --settings=settings.prod

# Update everything between Dec. 1, 2011 & Dec 31, 2011
./manage.py update_index --start='2011-12-01T00:00:00' --end='2011-12-31T23:59:59' --settings=settings.prod

# Update just a couple apps.
./manage.py update_index blog auth comments --settings=settings.prod

# Update just a single model (in a complex app).
./manage.py update_index auth.User --settings=settings.prod

# Crazy Go-Nuts University
./manage.py update_index events.Event media news.Story --start='2011-01-01T00:00:00' --remove --using=
```

Note: This command *ONLY* updates records in the index. It does *NOT* handle deletions unless the `--remove` flag is provided. You might consider a queue consumer if the memory requirements for `--remove` don't fit your needs. Alternatively, you can use the `RealtimeSignalProcessor`, which will automatically handle deletions.

1.5.3 rebuild_index

A shortcut for `clear_index` followed by `update_index`. It accepts any/all of the arguments of the following arguments:

```

`--age`:
    Number of hours back to consider objects new. Useful for nightly
    reindexes (`--age=24`). Requires `SearchIndexes` to implement
    the `get_updated_field` method.
`--batch-size`:
    Number of items to index at once. Default is 1000.
`--site`:
    The site object to use when reindexing (like `search_sites.mysite`).
`--noinput`:
    If provided, the interactive prompts are skipped and the index is
    uncerimoniously wiped out.
`--remove`:
    Remove objects from the index that are no longer present in the
    database.
`--verbosity`:
    If provided, dumps out more information about what's being done.

    * `0` = No output
    * `1` = Minimal output describing what models were indexed
      and how many records.
    * `2` = Full output, including everything from `1` plus output
      on each batch that is indexed, which is useful when debugging.
`--using`:
    If provided, determines which connection should be used. Default is
    `default`.

```

For when you really, really want a completely rebuilt index.

1.5.4 build_solr_schema

Once all of your `SearchIndex` classes are in place, this command can be used to generate the XML schema Solr needs to handle the search data. It accepts the following arguments:

```

`--filename`:
    If provided, directs output to a file instead of stdout.
`--using`:
    If provided, determines which connection should be used. Default is
    `default`.

```

Warning: This command does NOT update the `schema.xml` file for you. You either have to specify a filename flag or have to copy-paste (or redirect) the output to the correct file. Haystack has no way of knowing where your Solr is setup (or if it's even on the same machine), hence the manual step.

1.5.5 haystack_info

Provides some basic information about how Haystack is setup and what models it is handling. It accepts no arguments. Useful when debugging or when using Haystack-enabled third-party apps.

1.6 (In)Frequently Asked Questions

1.6.1 What is Haystack?

Haystack is meant to be a portable interface to a search engine of your choice. Some might call it a search framework, an abstraction layer or what have you. The idea is that you write your search code once and should be able to freely switch between backends as your situation necessitates.

1.6.2 Why should I consider using Haystack?

Haystack is targeted at the following use cases:

- If you want to feature search on your site and search solutions like Google or Yahoo search don't fit your needs.
- If you want to be able to customize your search and search on more than just the main content.
- If you want to have features like drill-down (faceting) or "More Like This".
- If you want a interface that is non-search engine specific, allowing you to change your mind later without much rewriting.

1.6.3 When should I not be using Haystack?

- Non-Model-based data. If you just want to index random data (flat files, alternate sources, etc.), Haystack isn't a good solution. Haystack is very `Model`-based and doesn't work well outside of that use case.
- Ultra-high volume. Because of the very nature of Haystack (abstraction layer), there's more overhead involved. This makes it portable, but as with all abstraction layers, you lose a little performance. You also can't take full advantage of the exact feature-set of your search engine. This is the price of pluggable backends.

1.6.4 Why was Haystack created when there are so many other search options?

The proliferation of search options in Django is a relatively recent development and is actually one of the reasons for Haystack's existence. There are too many options that are only partial solutions or are too engine specific.

Further, most use an unfamiliar API and documentation is lacking in most cases.

Haystack is an attempt to unify these efforts into one solution. That's not to say there should be no alternatives, but Haystack should provide a good solution to 80%+ of the search use cases out there.

1.6.5 What's the history behind Haystack?

Haystack started because of my frustration with the lack of good search options (before many other apps came out) and as the result of extensive use of DjangoSearch. DjangoSearch was a decent solution but had a number of shortcomings, such as:

- Tied to the `models.py`, so you'd have to modify the source of third-party (or `django.contrib`) apps in order to effectively use it.
- All or nothing approach to indexes. So all indexes appear on all sites and in all places.
- Lack of tests.
- Lack of documentation.
- Uneven backend implementations.

The initial idea was to simply fork Djangosearch and improve on these (and other issues). However, after stepping back, I decided to overhaul the entire API (and most of the underlying code) to be more representative of what I would want as an end-user. The result was starting afresh and reusing concepts (and some code) from Djangosearch as needed.

As a result of this heritage, you can actually still find some portions of Djangosearch present in Haystack (especially in the `SearchIndex` and `SearchBackend` classes) where it made sense. The original authors of Djangosearch are aware of this and thus far have seemed to be fine with this reuse.

1.6.6 Why doesn't <search engine X> have a backend included in Haystack?

Several possibilities on this.

1. Licensing

A common problem is that the Python bindings for a specific engine may have been released under an incompatible license. The goal is for Haystack to remain BSD licensed and importing bindings with an incompatible license can technically convert the entire codebase to that license. This most commonly occurs with GPL'ed bindings.

2. Lack of time

The search engine in question may be on the list of backends to add and we simply haven't gotten to it yet. We welcome patches for additional backends.

3. Incompatible API

In order for an engine to work well with Haystack, a certain baseline set of features is needed. This is often an issue when the engine doesn't support ranged queries or additional attributes associated with a search record.

4. We're not aware of the engine

If you think we may not be aware of the engine you'd like, please tell us about it (preferably via the group - <http://groups.google.com/group/django-haystack/>). Be sure to check through the backends (in case it wasn't documented) and search the history on the group to minimize duplicates.

1.7 Sites Using Haystack

The following sites are a partial list of people using Haystack. I'm always interested in adding more sites, so please find me (daniellindsley) via IRC or the mailing list thread.

1.7.1 LJWorld/Lawrence.com/KUSports

For all things search-related.

Using: Solr

- <http://www2.ljworld.com/search/>
- <http://www2.ljworld.com/search/vertical/news.story/>
- <http://www2.ljworld.com/marketplace/>
- <http://www.lawrence.com/search/>
- <http://www.kusports.com/search/>

1.7.2 AltWeeklies

Providing an API to story aggregation.

Using: Whoosh

- <http://www.northcoastjournal.com/altweeklies/documentation/>

1.7.3 Trapeze

Various projects.

Using: Xapian

- <http://www.trapeze.com/>
- <http://www.windmobile.ca/>
- <http://www.bonefishgrill.com/>
- <http://www.canadiantire.ca/> (Portions of)

1.7.4 Vickerey.com

For (really well done) search & faceting.

Using: Solr

- <http://store.vickerey.com/products/search/>

1.7.5 Eldarion

Various projects.

Using: Solr

- <http://eldarion.com/>

1.7.6 Sunlight Labs

For general search.

Using: Whoosh & Solr

- <http://sunlightlabs.com/>
- <http://subsidyscope.com/>

1.7.7 NASA

For general search.

Using: Solr

- An internal site called SMD Spacebook 1.1.
- <http://science.nasa.gov/>

1.7.8 AllForLocal

For general search.

- <http://www.allforlocal.com/>

1.7.9 HUGE

Various projects.

Using: Solr

- <http://hugeinc.com/>
- <http://houselogic.com/>

1.7.10 Brick Design

For search on Explore.

Using: Solr

- <http://bricksf.com/>
- <http://explore.org/>

1.7.11 Winding Road

For general search.

Using: Solr

- <http://www.windingroad.com/>

1.7.12 Reddit

For Reddit Gifts.

Using: Whoosh

- <http://redditgifts.com/>

1.7.13 Pegasus News

For general search.

Using: Xpian

- <http://www.pegasusnews.com/>

1.7.14 Rampframe

For general search.

Using: Xapian

- <http://www.rampframe.com/>

1.7.15 Forkinit

For general search, model-specific search and suggestions via MLT.

Using: Solr

- <http://forkinit.com/>

1.7.16 Structured Abstraction

For general search.

Using: Xapian

- <http://www.structuredabstraction.com/>
- <http://www.delivergood.org/>

1.7.17 CustomMade

For general search.

Using: Solr

- <http://www.custommade.com/>

1.7.18 University of the Andes, Dept. of Political Science

For general search & section-specific search. Developed by Monoku.

Using: Solr

- <http://www.congresovisible.org/>
- <http://www.monoku.com/>

1.7.19 Christchurch Art Gallery

For general search & section-specific search.

Using: Solr

- <http://christchurchartgallery.org.nz/search/>
- <http://christchurchartgallery.org.nz/collection/browse/>

1.7.20 DevCheatSheet.com

For general search.

Using: Xapian

- <http://devcheatsheet.com/>

1.7.21 TodasLasRecetas

For search, faceting & More Like This.

Using: Solr

- <http://www.todaslasrecetas.es/receta/s?q=langostinos>
- <http://www.todaslasrecetas.es/receta/9526/brochetas-de-langostinos>

1.7.22 AstroBin

For general search.

Using: Solr

- <http://www.astrobin.com/>

1.7.23 European Paper Company

For general search.

Using: ???

- <http://europeanpaper.com/>

1.7.24 mtn-op

For general search.

Using: ???

- <http://mountain-op.com/>

1.7.25 Crate

Crate is a PyPI mirror/replacement. It's using Haystack to power all search & faceted navigation on the site.

Using: Elasticsearch

- <https://crate.io/>

1.7.26 Pix Populi

Pix Populi is a popular French photo sharing site.

Using: Solr

- <http://www.pix-populi.fr/>

1.7.27 LocalWiki

LocalWiki is a tool for collaborating in local, geographic communities. It's using Haystack to power search on every LocalWiki instance.

Using: Solr

- <http://localwiki.org/>

1.7.28 Pitchup

For faceting, geo and autocomplete.

Using: ???

- <http://www.pitchup.com/search/>

1.7.29 Gidsy

Gidsy makes it easy for anyone to organize and find exciting things to do everywhere in the world.

For activity search, area pages, forums and private messages.

Using: Elasticsearch

- <https://gidsy.com/>
- <https://gidsy.com/search/>
- <https://gidsy.com/forum/>

1.8 Haystack-Related Applications

1.8.1 Sub Apps

These are apps that build on top of the infrastructure provided by Haystack. Useful for essentially extending what Haystack can do.

queued_search

http://github.com/toastdriven/queued_search (2.X compatible)

Provides a queue-based setup as an alternative to `RealtimeSignalProcessor` or constantly running the `update_index` command. Useful for high-load, short update time situations.

celery-haystack

<https://github.com/jezdez/celery-haystack> (1.X and 2.X compatible)

Also provides a queue-based setup, this time centered around Celery. Useful for keeping the index fresh per model instance or with the included task to call the `update_index` management command instead.

django-celery-haystack

<https://github.com/mixcloud/django-celery-haystack-SearchIndex>

Another queue-based setup, also around Celery. Useful for keeping the index fresh.

saved_searches

http://github.com/toastdriven/saved_searches (2.X compatible)

Adds personalization to search. Retains a history of queries run by the various users on the site (including anonymous users). This can be used to present the user with their search history and provide most popular/most recent queries on the site.

saved-search

<https://github.com/DirectEmployers/saved-search>

An alternate take on persisting user searches, this has a stronger focus on locale-based searches as well as further integration.

haystack-static-pages

<http://github.com/trapeze/haystack-static-pages>

Provides a simple way to index flat (non-model-based) content on your site. By using the management command that comes with it, it can crawl all pertinent pages on your site and add them to search.

django-tumbleweed

<http://github.com/mcroydon/django-tumbleweed>

Provides a tumblelog-like view to any/all Haystack-enabled models on your site. Useful for presenting date-based views of search data. Attempts to avoid the database completely where possible.

1.8.2 Haystack-Enabled Apps

These are reusable apps that ship with `SearchIndexes`, suitable for quick integration with Haystack.

- `django-faq` (freq. asked questions app) - <http://github.com/benspaulding/django-faq>
- `django-essays` (blog-like essay app) - <http://github.com/bkeating/django-essays>
- `gtalug` (variety of apps) - <http://github.com/myles/gtalug>
- `sciencemuseum` (science museum open data) - <http://github.com/simonw/sciencemuseum>
- `vz-wiki` (wiki) - <http://github.com/jobscry/vz-wiki>

- `ffmff` (events app) - <http://github.com/stefreak/ffmff>
- `Dinette` (forums app) - <http://github.com/uswaretech/Dinette>
- `fiftystates_site` (site) - http://github.com/sunlightlabs/fiftystates_site
- `Open-Knesset` (site) - <http://github.com/ofri/Open-Knesset>

1.9 Installing Search Engines

1.9.1 Solr

Official Download Location: <http://www.apache.org/dyn/closer.cgi/lucene/solr/>

Solr is Java but comes in a pre-packaged form that requires very little other than the JRE and Jetty. It's very performant and has an advanced featureset. Haystack suggests using Solr 3.5+, though it's possible to get it working on Solr 1.4 with a little effort. Installation is relatively simple:

```
curl -O http://apache.mirrors.tds.net/lucene/solr/3.5.0/apache-solr-3.5.0.tgz
tar xvzf apache-solr-3.5.0.tgz
cd apache-solr-3.5.0
cd example
java -jar start.jar
```

You'll need to revise your schema. You can generate this from your application (once Haystack is installed and setup) by running `./manage.py build_solr_schema`. Take the output from that command and place it in `apache-solr-3.5.0/example/solr/conf/schema.xml`. Then restart Solr.

Note: `build_solr_schema` uses a template to generate `schema.xml`. Haystack provides a default template using some sensible defaults. If you would like to provide your own template, you will need to place it in `search_configuration/solr.xml`, inside a directory specified by your app's `TEMPLATE_DIRS` setting. Examples:

```
/myproj/myapp/templates/search_configuration/solr.xml
# ...or...
/myproj/templates/search_configuration/solr.xml
```

You'll also need a Solr binding, `pysolr`. The official `pysolr` package, distributed via PyPI, is the best version to use (2.1.0+). Place `pysolr.py` somewhere on your `PYTHONPATH`.

Note: `pysolr` has it's own dependencies that aren't covered by Haystack. For best results, you should have an `ElementTree` variant install (preferably the `lxml` variant), `httplib2` for timeouts (though it will fall back to `httplib`) and either the `json` module that comes with Python 2.5+ or `simplejson`.

More Like This

To enable the "More Like This" functionality in Haystack, you'll need to enable the `MoreLikeThisHandler`. Add the following line to your `solrconfig.xml` file within the `config` tag:

```
<requestHandler name="/mlt" class="solr.MoreLikeThisHandler" />
```

Spelling Suggestions

To enable the spelling suggestion functionality in Haystack, you'll need to enable the `SpellCheckComponent`.

The first thing to do is create a special field on your `SearchIndex` class that mirrors the `text` field, but uses `FacetCharField`. This disables the post-processing that Solr does, which can mess up your suggestions. Something like the following is suggested:

```
class MySearchIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    # ... normal fields then...
    suggestions = indexes.FacetCharField()

    def prepare(self, obj):
        prepared_data = super(NoteIndex, self).prepare(obj)
        prepared_data['suggestions'] = prepared_data['text']
        return prepared_data
```

Then, you enable it in Solr by adding the following line to your `solrconfig.xml` file within the `config` tag:

```
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">

  <str name="queryAnalyzerFieldType">textSpell</str>

  <lst name="spellchecker">
    <str name="name">default</str>
    <str name="field">suggestions</str>
    <str name="spellcheckIndexDir">./spellchecker1</str>
    <str name="buildOnCommit">>true</str>
  </lst>
</searchComponent>
```

Then change your default handler from:

```
<requestHandler name="standard" class="solr.StandardRequestHandler" default="true" />
```

... to ...:

```
<requestHandler name="standard" class="solr.StandardRequestHandler" default="true">
  <arr name="last-components">
    <str>spellcheck</str>
  </arr>
</requestHandler>
```

Be warned that the `<str name="field">suggestions</str>` portion will be specific to your `SearchIndex` classes (in this case, assuming the main field is called `text`).

1.9.2 Elasticsearch

Official Download Location: <http://www.elasticsearch.org/download/>

Elasticsearch is Java but comes in a pre-packaged form that requires very little other than the JRE. It's also very performant, scales easily and has an advanced featureset. Haystack requires at least version 0.17.7 (0.18.6 is current as of writing). Installation is best done using a package manager:

```
# On Mac OS X...
brew install elasticsearch

# On Ubuntu...
```

```
apt-get install elasticsearch

# Then start via:
elasticsearch -f -D es.config=<path to YAML config>

# Example:
elasticsearch -f -D es.config=/usr/local/Cellar/elasticsearch/0.17.7/config/elasticsearch.yml
```

You may have to alter the configuration to run on localhost when developing locally. Modifications should be done in a YAML file, the stock one being `config/elasticsearch.yml`:

```
# Unicast Discovery (disable multicast)
discovery.zen.ping.multicast.enabled: false
discovery.zen.ping.unicast.hosts: ["127.0.0.1"]

# Name your cluster here to whatever.
# My machine is called "Venus", so...
cluster:
  name: venus

network:
  host: 127.0.0.1

path:
  logs: /usr/local/var/log
  data: /usr/local/var/data
```

You'll also need an Elasticsearch binding: `pyelasticsearch` (**NOT** `pyes`). Place `pyelasticsearch` somewhere on your `PYTHONPATH` (usually `python setup.py install` or `pip install pyelasticsearch`).

Note: `pyelasticsearch` has its own dependencies that aren't covered by Haystack. You'll also need `requests` & `simplejson` for speedier JSON construction/parsing.

1.9.3 Whoosh

Official Download Location: <http://bitbucket.org/mchaput/whoosh/>

Whoosh is pure Python, so it's a great option for getting started quickly and for development, though it does work for small scale live deployments. The current recommended version is 1.3.1+. You can install via `PyPI` using:

```
sudo easy_install whoosh
# ... or ...
sudo pip install whoosh
```

Note that, while capable otherwise, the Whoosh backend does not currently support "More Like This" or faceting. Support for these features has recently been added to Whoosh itself & may be present in a future release.

1.9.4 Xapian

Official Download Location: <http://xapian.org/download>

Xapian is written in C++ so it requires compilation (unless your OS has a package for it). Installation looks like:


```
curl -O http://oligarchy.co.uk/xapian/1.0.11/xapian-core-1.0.11.tar.gz
curl -O http://oligarchy.co.uk/xapian/1.0.11/xapian-bindings-1.0.11.tar.gz

tar xvzf xapian-core-1.0.11.tar.gz
tar xvzf xapian-bindings-1.0.11.tar.gz

cd xapian-core-1.0.11
./configure
make
sudo make install

cd ..
cd xapian-bindings-1.0.11
./configure
make
sudo make install
```

Xapian is a third-party supported backend. It is not included in Haystack proper due to licensing. To use it, you need both Haystack itself as well as `xapian-haystack`. You can download the source from <http://github.com/notanumber/xapian-haystack/tree/master>. Installation instructions can be found on that page as well. The backend, written by David Sauve (notanumber), fully implements the *SearchQuerySet* API and is an excellent alternative to Solr.

1.10 Debugging Haystack

There are some common problems people run into when using Haystack for the first time. Some of the common problems and things to try appear below.

Note: As a general suggestion, your best friend when debugging an issue is to use the `pdb` library included with Python. By dropping a `import pdb; pdb.set_trace()` in your code before the issue occurs, you can step through and examine variable/logic as you progress through. Make sure you don't commit those `pdb` lines though.

1.10.1 “No module named haystack.”

This problem usually occurs when first adding Haystack to your project.

- Are you using the `haystack` directory within your `django-haystack` checkout/install?
- Is the `haystack` directory on your `PYTHONPATH`? Alternatively, is `haystack` symlinked into your project?
- Start a Django shell (`./manage.py shell`) and try `import haystack`. You may receive a different, more descriptive error message.
- Double-check to ensure you have no circular imports. (i.e. module A tries importing from module B which is trying to import from module A.)

1.10.2 “No results found.” (On the web page)

Several issues can cause no results to be found. Most commonly it is either not running a `rebuild_index` to populate your index or having a blank `document=True` field, resulting in no content for the engine to search on.

- Do you have a `search_indexes.py` located within an installed app?

- Do you have data in your database?
- Have you run a `./manage.py rebuild_index` to index all of your content?
- Try running `./manage.py rebuild_index -v2` for more verbose output to ensure data is being processed/inserted.
- Start a Django shell (`./manage.py shell`) and try:

```
>>> from haystack.query import SearchQuerySet
>>> sqs = SearchQuerySet().all()
>>> sqs.count()
```

- You should get back an integer > 0 . If not, check the above and reindex.

```
>>> sqs[0] # Should get back a SearchResult object.
>>> sqs[0].id # Should get something back like 'myapp.mymodel.1'.
>>> sqs[0].text # ... or whatever your document=True field is.
```

- If you get back either `u''` or `None`, it means that your data isn't making it into the main field that gets searched. You need to check that the field either has a template that uses the model data, a `model_attr` that pulls data directly from the model or a `prepare/prepare_FOO` method that populates the data at index time.
- Check the template for your search page and ensure it is looping over the results properly. Also ensure that it's either accessing valid fields coming back from the search engine or that it's trying to access the associated model via the `{{ result.object.foo }}` lookup.

1.10.3 “LockError: [Errno 17] File exists: '/path/to/whoosh_index/_MAIN_LOCK'”

This is a Whoosh-specific traceback. It occurs when the Whoosh engine in one process/thread locks the index files for writing while another process/thread tries to access them. This is a common error when using `RealtimeSignalProcessor` with Whoosh under any kind of load, which is why it's only recommended for small sites or development.

The only real solution is to set up a cron job that runs `./manage.py rebuild_index` (optionally with `--age=24`) that runs nightly (or however often you need) to refresh the search indexes. Then disable the use of the `RealtimeSignalProcessor` within your settings.

The downside to this is that you lose real-time search. For many people, this isn't an issue and this will allow you to scale Whoosh up to a much higher traffic. If this is not acceptable, you should investigate either the Solr or Xapian backends.

1.10.4 “Failed to add documents to Solr: [Reason: None]”

This is a Solr-specific traceback. It generally occurs when there is an error with your `HAYSTACK_CONNECTIONS[<alias>]['URL']`. Since Solr acts as a webservice, you should test the URL in your web browser. If you receive an error, you may need to change your URL.

This can also be caused when using old versions of `pysolr` (2.0.9 and before), using `httplib2` and including a trailing slash in your `HAYSTACK_CONNECTIONS[<alias>]['URL']`. Please upgrade your version of `pysolr` (2.0.13+).

1.10.5 “Got an unexpected keyword argument ‘boost’”

This is a Solr-specific traceback. This can also be caused when using old versions of `pysolr` (2.0.12 and before). Please upgrade your version of `pysolr` (2.0.13+).

1.11 Migrating From Haystack 1.X to Haystack 2.X

Haystack introduced several backward-incompatible changes in the process of moving from the 1.X series to the 2.X series. These were done to clean up the API, to support new features & to clean up problems in 1.X. At a high level, they consisted of:

- The removal of `SearchSite` & `haystack.site`.
- The removal of `handle_registrations` & `autodiscover`.
- The addition of multiple index support.
- The addition of `SignalProcessors` & the removal of `RealTimeSearchIndex`.
- The removal/renaming of various settings.

This guide will help you make the changes needed to be compatible with Haystack 2.X.

1.11.1 Settings

Most prominently, the old way of specifying a backend & its settings has changed to support the multiple index feature. A complete Haystack 1.X example might look like:

```
HAYSTACK_SEARCH_ENGINE = 'solr'
HAYSTACK_SOLR_URL = 'http://localhost:9001/solr/default'
HAYSTACK_SOLR_TIMEOUT = 60 * 5
HAYSTACK_INCLUDE_SPELLING = True
HAYSTACK_BATCH_SIZE = 100

# Or...
HAYSTACK_SEARCH_ENGINE = 'whoosh'
HAYSTACK_WHOOSH_PATH = '/home/search/whoosh_index'
HAYSTACK_WHOOSH_STORAGE = 'file'
HAYSTACK_WHOOSH_POST_LIMIT = 128 * 1024 * 1024
HAYSTACK_INCLUDE_SPELLING = True
HAYSTACK_BATCH_SIZE = 100

# Or...
HAYSTACK_SEARCH_ENGINE = 'xapian'
HAYSTACK_XAPIAN_PATH = '/home/search/xapian_index'
HAYSTACK_INCLUDE_SPELLING = True
HAYSTACK_BATCH_SIZE = 100
```

In Haystack 2.X, you can now supply as many backends as you like, so all of the above settings can now be active at the same time. A translated set of settings would look like:

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.solr_backend.SolrEngine',
        'URL': 'http://localhost:9001/solr/default',
        'TIMEOUT': 60 * 5,
        'INCLUDE_SPELLING': True,
        'BATCH_SIZE': 100,
    },
    'autocomplete': {
        'ENGINE': 'haystack.backends.whoosh_backend.WhooshEngine',
        'PATH': '/home/search/whoosh_index',
        'STORAGE': 'file',
        'POST_LIMIT': 128 * 1024 * 1024,
```

```
'INCLUDE_SPELLING': True,
'BATCH_SIZE': 100,
},
'slave': {
  'ENGINE': 'xapian_backend.XapianEngine',
  'PATH': '/home/search/xapian_index',
  'INCLUDE_SPELLING': True,
  'BATCH_SIZE': 100,
},
}
```

You are required to have at least one connection listed within `HAYSTACK_CONNECTIONS`, it must be named `default` & it must have a valid `ENGINE` within it. Bare minimum looks like:

```
HAYSTACK_CONNECTIONS = {
  'default': {
    'ENGINE': 'haystack.backends.simple_backend.SimpleEngine'
  }
}
```

The key for each backend is an identifier you use to describe the backend within your app. You should refer to the [Multiple Indexes](#) documentation for more information on using the new multiple indexes & routing features.

Also note that the `ENGINE` setting has changed from a lowercase “short name” of the engine to a full path to a new Engine class within the backend. Available options are:

- `haystack.backends.solr_backend.SolrEngine`
- `haystack.backends.whoosh_backend.WhooshEngine`
- `haystack.backends.simple_backend.SimpleEngine`

Additionally, the following settings were outright removed & will generate an exception if found:

- `HAYSTACK_SITECONF` - Remove this setting & the file it pointed to.
- `HAYSTACK_ENABLE_REGISTRATIONS`
- `HAYSTACK_INCLUDE_SPELLING`

1.11.2 Backends

The dummy backend was outright removed from Haystack, as it served very little use after the `simple` (pure-ORM-powered) backend was introduced.

If you wrote a custom backend, please refer to the “Custom Backends” section below.

1.11.3 Indexes

The other major changes affect the `SearchIndex` class. As the concept of `haystack.site` & `SearchSite` are gone, you’ll need to modify your indexes.

A Haystack 1.X index might’ve looked like:

```
import datetime
from haystack.indexes import *
from haystack import site
from myapp.models import Note
```

```

class NoteIndex(SearchIndex):
    text = CharField(document=True, use_template=True)
    author = CharField(model_attr='user')
    pub_date = DateTimeField(model_attr='pub_date')

    def get_queryset(self):
        """Used when the entire index for model is updated."""
        return Note.objects.filter(pub_date__lte=datetime.datetime.now())

site.register(Note, NoteIndex)

```

A converted Haystack 2.X index should look like:

```

import datetime
from haystack import indexes
from myapp.models import Note

class NoteIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    author = indexes.CharField(model_attr='user')
    pub_date = indexes.DateTimeField(model_attr='pub_date')

    def get_model(self):
        return Note

    def index_queryset(self, using=None):
        """Used when the entire index for model is updated."""
        return self.get_model().objects.filter(pub_date__lte=datetime.datetime.now())

```

Note the import on `site` & the registration statements are gone. Newly added are is the `NoteIndex.get_model` method. This is a **required** method & should simply return the `Model` class the index is for.

There's also a new, additional class added to the class definition. The `indexes.Indexable` class is a simple mixin that serves to identify the classes Haystack should automatically discover & use. If you have a custom base class (say `QueuedSearchIndex`) that other indexes inherit from, simply leave the `indexes.Indexable` off that declaration & Haystack won't try to use it.

Additionally, the name of the `document=True` field is now enforced to be `text` across all indexes. If you need it named something else, you should set the `HAYSTACK_DOCUMENT_FIELD` setting. For example:

```
HAYSTACK_DOCUMENT_FIELD = 'pink_polka_dot'
```

Finally, the `index_queryset` method should supplant the `get_queryset` method. This was present in the Haystack 1.2.X series (with a deprecation warning in 1.2.4+) but has been removed in Haystack v2.

Finally, if you were unregistering other indexes before, you should make use of the new `EXCLUDED_INDEXES` setting available in each backend's settings. It should be a list of strings that contain the Python import path to the indexes that should not be loaded & used. For example:

```

HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.solr_backend.SolrEngine',
        'URL': 'http://localhost:9001/solr/default',
        'EXCLUDED_INDEXES': [
            # Imagine that these indexes exist. They don't.
            'django.contrib.auth.search_indexes.UserIndex',
            'third_party_blog_app.search_indexes.EntryIndex',

```

```
    ]
  }
}
```

This allows for reliable swapping of the index that handles a model without relying on correct import order.

1.11.4 Removal of `RealTimeSearchIndex`

Use of the `haystack.indexes.RealTimeSearchIndex` is no longer valid. It has been removed in favor of `RealtimeSignalProcessor`. To migrate, first change the inheritance of all your `RealTimeSearchIndex` subclasses to use `SearchIndex` instead:

```
# Old.
class MySearchIndex(indexes.RealTimeSearchIndex, indexes.Indexable):
    # ...

# New.
class MySearchIndex(indexes.SearchIndex, indexes.Indexable):
    # ...
```

Then update your settings to enable use of the `RealtimeSignalProcessor`:

```
HAYSTACK_SIGNAL_PROCESSOR = 'haystack.signals.RealtimeSignalProcessor'
```

1.11.5 Done!

For most basic uses of Haystack, this is all that is necessary to work with Haystack 2.X. You should rebuild your index if needed & test your new setup.

1.11.6 Advanced Uses

Swapping Backend

If you were manually swapping the `SearchQuery` or `SearchBackend` being used by `SearchQuerySet` in the past, it's now preferable to simply setup another connection & use the `SearchQuerySet.using` method to select that connection instead.

Also, if you were manually instantiating `SearchBackend` or `SearchQuery`, it's now preferable to rely on the connection's engine to return the right thing. For example:

```
from haystack import connections
backend = connections['default'].get_backend()
query = connections['default'].get_query()
```

Custom Backends

If you had written a custom `SearchBackend` and/or custom `SearchQuery`, there's a little more work needed to be Haystack 2.X compatible.

You should, but don't have to, rename your `SearchBackend` & `SearchQuery` classes to be more descriptive/less collide-y. For example, `solr_backend.SearchBackend` became `solr_backend.SolrSearchBackend`. This prevents non-namespaced imports from stomping on each other.

You need to add a new class to your backend, subclassing `BaseEngine`. This allows specifying what backend & query should be used on a connection with less duplication/naming trickery. It goes at the bottom of the file (so that the classes are defined above it) and should look like:

```
from haystack.backends import BaseEngine
from haystack.backends.solr_backend import SolrSearchQuery

# Code then...

class MyCustomSolrEngine(BaseEngine):
    # Use our custom backend.
    backend = MySolrBackend
    # Use the built-in Solr query.
    query = SolrSearchQuery
```

Your `HAYSTACK_CONNECTIONS['default']['ENGINE']` should then point to the full Python import path to your new `BaseEngine` subclass.

Finally, you will likely have to adjust the `SearchBackend.__init__` & `SearchQuery.__init__`, as they have changed significantly. Please refer to the commits for those backends.

1.12 Contributing

Haystack is open-source and, as such, grows (or shrinks) & improves in part due to the community. Below are some guidelines on how to help with the project.

1.12.1 Philosophy

- Haystack is BSD-licensed. All contributed code must be either
 - the original work of the author, contributed under the BSD, or...
 - work taken from another project released under a BSD-compatible license.
- GPL'd (or similar) works are not eligible for inclusion.
- Haystack's git master branch should always be stable, production-ready & passing all tests.
- Major releases (1.x.x) are commitments to backward-compatibility of the public APIs. Any documented API should ideally not change between major releases. The exclusion to this rule is in the event of either a security issue or to accommodate changes in Django itself.
- Minor releases (x.3.x) are for the addition of substantial features or major bugfixes.
- Patch releases (x.x.4) are for minor features or bugfixes.

1.12.2 Guidelines For Reporting An Issue/Feature

So you've found a bug or have a great idea for a feature. Here's the steps you should take to help get it added/fixd in Haystack:

- First, check to see if there's an existing issue/pull request for the bug/feature. All issues are at <https://github.com/toastdriven/django-haystack/issues> and pull reqs are at <https://github.com/toastdriven/django-haystack/pulls>.
- If there isn't one there, please file an issue. The ideal report includes:

- A description of the problem/suggestion.
- How to recreate the bug.
- If relevant, including the versions of your:
 - * Python interpreter
 - * Django
 - * Haystack
 - * Search engine used (as well as bindings)
 - * Optionally of the other dependencies involved
- Ideally, creating a pull request with a (failing) test case demonstrating what’s wrong. This makes it easy for us to reproduce & fix the problem. Instructions for running the tests are at [Welcome to Haystack!](#)

You might also hop into the IRC channel (`#haystack` on `irc.freenode.net`) & raise your question there, as there may be someone who can help you with a work-around.

1.12.3 Guidelines For Contributing Code

If you’re ready to take the plunge & contribute back some code/docs, the process should look like:

- Fork the project on GitHub into your own account.
- Clone your copy of Haystack.
- Make a new branch in git & commit your changes there.
- Push your new branch up to GitHub.
- Again, ensure there isn’t already an issue or pull request out there on it. If there is & you feel you have a better fix, please take note of the issue number & mention it in your pull request.
- Create a new pull request (based on your branch), including what the problem/feature is, versions of your software & referencing any related issues/pull requests.

In order to be merged into Haystack, contributions must have the following:

- A solid patch that:
 - is clear.
 - works across all supported versions of Python/Django.
 - follows the existing style of the code base (mostly PEP-8).
 - comments included as needed.
- A test case that demonstrates the previous flaw that now passes with the included patch.
- If it adds/changes a public API, it must also include documentation for those changes.
- Must be appropriately licensed (see “Philosophy”).
- Adds yourself to the AUTHORS file.

If your contribution lacks any of these things, they will have to be added by a core contributor before being merged into Haystack proper, which may take substantial time for the all-volunteer team to get to.

1.12.4 Guidelines For Core Contributors

If you've been granted the commit bit, here's how to shepherd the changes in:

- Any time you go to work on Haystack, please use `git pull --rebase` to fetch the latest changes.
- Any new features/bug fixes must meet the above guidelines for contributing code (solid patch/tests passing/docs included).
- Commits are typically cherry-picked onto a branch off master.
 - This is done so as not to include extraneous commits, as some people submit pull reqs based on their git master that has other things applied to it.
- A set of commits should be squashed down to a single commit.
 - `git merge --squash` is a good tool for performing this, as is `git rebase -i HEAD~N`.
 - This is done to prevent anyone using the git repo from accidentally pulling work-in-progress commits.
- Commit messages should use past tense, describe what changed & thank anyone involved. Examples:

```

"""Added support for the latest version of Whoosh (v2.3.2)."""
"""Fixed a bug in ``solr_backend.py``. Thanks to joeschmoe for the report!"""
"""BACKWARD-INCOMPATIBLE: Altered the arguments passed to ``SearchBackend``.

Further description appears here if the change warrants an explanation
as to why it was done."""

```

- For any patches applied from a contributor, please ensure their name appears in the AUTHORS file.
- When closing issues or pull requests, please reference the SHA in the closing message (i.e. Thanks! Fixed in SHA: 6b93f6). GitHub will automatically link to it.

Advanced Uses

Once you've got Haystack working, here are some of the more complex features you may want to include in your application.

2.1 Best Practices

What follows are some general recommendations on how to improve your search. Some tips represent performance benefits, some provide a better search index. You should evaluate these options for yourself and pick the ones that will work best for you. Not all situations are created equal and many of these options could be considered mandatory in some cases and unnecessary premature optimizations in others. Your mileage may vary.

2.1.1 Good Search Needs Good Content

Most search engines work best when they're given corpuses with predominantly text (as opposed to other data like dates, numbers, etc.) in decent quantities (more than a couple words). This is in stark contrast to the databases most people are used to, which rely heavily on non-text data to create relationships and for ease of querying.

To this end, if search is important to you, you should take the time to carefully craft your `SearchIndex` subclasses to give the search engine the best information you can. This isn't necessarily hard but is worth the investment of time and thought. Assuming you've only ever used the `BasicSearchIndex`, in creating custom `SearchIndex` classes, there are some easy improvements to make that will make your search better:

- For your `document=True` field, use a well-constructed template.
- Add fields for data you might want to be able to filter by.
- If the model has related data, you can squash good content from those related models into the parent model's `SearchIndex`.
- Similarly, if you have heavily de-normalized models, it may be best represented by a single indexed model rather than many indexed models.

Well-Constructed Templates

A relatively unique concept in Haystack is the use of templates associated with `SearchIndex` fields. These are data templates, will never be seen by users and ideally contain no HTML. They are used to collect various data from the model and structure it as a document for the search engine to analyze and index.

Note: If you read nothing else, this is the single most important thing you can do to make search on your site better for your users. Good templates can make or break your search and providing the search engine with good content to index is critical.

Good templates structure the data well and incorporate as much pertinent text as possible. This may include additional fields such as titles, author information, metadata, tags/categories. Without being artificial, you want to construct as much context as you can. This doesn't mean you should necessarily include every field, but you should include fields that provide good content or include terms you think your users may frequently search on.

Unless you have very unique numbers or dates, neither of these types of data are a good fit within templates. They are usually better suited to other fields for filtering within a `SearchQuerySet`.

Additional Fields For Filtering

Documents by themselves are good for generating indexes of content but are generally poor for filtering content, for instance, by date. All search engines supported by Haystack provide a means to associate extra data as attributes/fields on a record. The database analogy would be adding extra columns to the table for filtering.

Good candidates here are date fields, number fields, de-normalized data from related objects, etc. You can expose these things to users in the form of a calendar range to specify, an author to look up or only data from a certain series of numbers to return.

You will need to plan ahead and anticipate what you might need to filter on, though with each field you add, you increase storage space usage. It's generally **NOT** recommended to include every field from a model, just ones you are likely to use.

Related Data

Related data is somewhat problematic to deal with, as most search engines are better with documents than they are with relationships. One way to approach this is to de-normalize a related child object or objects into the parent's document template. The inclusion of a foreign key's relevant data or a simple Django `{% for %}` templatetag to iterate over the related objects can increase the salient data in your document. Be careful what you include and how you structure it, as this can have consequences on how well a result might rank in your search.

2.1.2 Avoid Hitting The Database

A very easy but effective thing you can do to drastically reduce hits on the database is to pre-render your search results using stored fields then disabling the `load_all` aspect of your `SearchView`.

Warning: This technique may cause a substantial increase in the size of your index as you are basically using it as a storage mechanism.

To do this, you setup one or more stored fields (`indexed=False`) on your `SearchIndex` classes. You should specify a template for the field, filling it with the data you'd want to display on your search results pages. When the model attached to the `SearchIndex` is placed in the index, this template will get rendered and stored in the index alongside the record.

Note: The downside of this method is that the HTML for the result will be locked in once it is indexed. To make changes to the structure, you'd have to reindex all of your content. It also limits you to a single display of the content (though you could use multiple fields if that suits your needs).

The second aspect is customizing your `SearchView` and its templates. First, pass the `load_all=False` to your `SearchView`, ideally in your `URLconf`. This prevents the `SearchQuerySet` from loading all models objects for results ahead of time. Then, in your template, simply display the stored content from your `SearchIndex` as the HTML result.

Warning: To do this, you must absolutely avoid using `{{ result.object }}` or any further accesses beyond that. That call will hit the database, not only nullifying your work on lessening database hits, but actually making it worse as there will now be at least query for each result, up from a single query for each type of model with `load_all=True`.

2.1.3 Content-Type Specific Templates

Frequently, when displaying results, you'll want to customize the HTML output based on what model the result represents.

In practice, the best way to handle this is through the use of `include` along with the data on the `SearchResult`.

Your existing loop might look something like:

```
{% for result in page.object_list %}
  <p>
    <a href="{{ result.object.get_absolute_url }}">{{ result.object.title }}</a>
  </p>
{% empty %}
  <p>No results found.</p>
{% endfor %}
```

An improved version might look like:

```
{% for result in page.object_list %}
  {% if result.content_type == "blog.post" %}
  {% include "search/includes/blog/post.html" %}
  {% endif %}
  {% if result.content_type == "media.photo" %}
  {% include "search/includes/media/photo.html" %}
  {% endif %}
{% empty %}
  <p>No results found.</p>
{% endfor %}
```

Those include files might look like:

```
# search/includes/blog/post.html
<div class="post_result">
  <h3><a href="{{ result.object.get_absolute_url }}">{{ result.object.title }}</a></h3>
  <p>{{ result.object.tease }}</p>
</div>

# search/includes/media/photo.html
<div class="photo_result">
  <a href="{{ result.object.get_absolute_url }}">
  </a>
  <p>Taken By {{ result.object.taken_by }}</p>
</div>
```

You can make this even better by standardizing on an includes layout, then writing a template tag or filter that generates the include filename. Usage might look something like:

```
{% for result in page.object_list %}
    {% with result|search_include as fragment %}
        {% include fragment %}
    {% endwith %}
{% empty %}
    <p>No results found.</p>
{% endfor %}
```

2.1.4 Real-Time Search

If your site sees heavy search traffic and up-to-date information is very important, Haystack provides a way to constantly keep your index up to date.

You can enable the `RealtimeSignalProcessor` within your settings, which will allow Haystack to automatically update the index whenever a model is saved/deleted.

You can find more information within the [Signal Processors](#) documentation.

2.1.5 Use Of A Queue For A Better User Experience

By default, you have to manually reindex content, Haystack immediately tries to merge it into the search index. If you have a write-heavy site, this could mean your search engine may spend most of its time churning on constant merges. If you can afford a small delay between when a model is saved and when it appears in the search results, queuing these merges is a good idea.

You gain a snappier interface for users as updates go into a queue (a fast operation) and then typical processing continues. You also get a lower churn rate, as most search engines deal with batches of updates better than many single updates. You can also use this to distribute load, as the queue consumer could live on a completely separate server from your webservers, allowing you to tune more efficiently.

Implementing this is relatively simple. There are two parts, creating a new `QueuedSignalProcessor` class and creating a queue processing script to handle the actual updates.

For the `QueuedSignalProcessor`, you should inherit from `haystack.signals.BaseSignalProcessor`, then alter the `setup/teardown` methods to call an `enqueueing` method instead of directly calling `handle_save/handle_delete`. For example:

```
from haystack import signals

class QueuedSignalProcessor(signals.BaseSignalProcessor):
    # Override the built-in.
    def setup(self):
        models.signals.post_save.connect(self.enqueue_save)
        models.signals.post_delete.connect(self.enqueue_delete)

    # Override the built-in.
    def teardown(self):
        models.signals.post_save.disconnect(self.enqueue_save)
        models.signals.post_delete.disconnect(self.enqueue_delete)

    # Add on a queuing method.
    def enqueue_save(self, sender, instance, **kwargs):
        # Push the save & information onto queue du jour here...

    # Add on a queuing method.
```

```
def enqueue_delete(self, sender, instance, **kwargs):
    # Push the delete & information onto queue du jour here...
```

For the consumer, this is much more specific to the queue used and your desired setup. At a minimum, you will need to periodically consume the queue, fetch the correct index from the `SearchSite` for your application, load the model from the message and pass that model to the `update_object` or `remove_object` methods on the `SearchIndex`. Proper grouping, batching and intelligent handling are all additional things that could be applied on top to further improve performance.

2.2 Highlighting

Haystack supports two different methods of highlighting. You can either use `SearchQuerySet.highlight` or the built-in `{% highlight %}` template tag, which uses the `Highlighter` class. Each approach has advantages and disadvantages you need to weigh when deciding which to use.

If you want portable, flexible, decently fast code, the `{% highlight %}` template tag (or manually using the underlying `Highlighter` class) is the way to go. On the other hand, if you care more about speed and will only ever be using one backend, `SearchQuerySet.highlight` may suit your needs better.

Use of `SearchQuerySet.highlight` is documented in the [SearchQuerySet API](#) documentation and the `{% highlight %}` tag is covered in the [Template Tags](#) documentation, so the rest of this material will cover the `Highlighter` implementation.

2.2.1 Highlighter

The `Highlighter` class is a pure-Python implementation included with Haystack that's designed for flexibility. If you use the `{% highlight %}` template tag, you'll be automatically using this class. You can also use it manually in your code. For example:

```
>>> from haystack.utils import Highlighter

>>> my_text = 'This is a sample block that would be more meaningful in real life.'
>>> my_query = 'block meaningful'

>>> highlight = Highlighter(my_query)
>>> highlight.highlight(my_text)
u'...<span class="highlighted">block</span> that would be more <span class="highlighted">meaningful</span>...'
```

The default implementation takes three optional kwargs: `html_tag`, `css_class` and `max_length`. These allow for basic customizations to the output, like so:

```
>>> from haystack.utils import Highlighter

>>> my_text = 'This is a sample block that would be more meaningful in real life.'
>>> my_query = 'block meaningful'

>>> highlight = Highlighter(my_query, html_tag='div', css_class='found', max_length=35)
>>> highlight.highlight(my_text)
u'...<div class="found">block</div> that would be more <div class="found">meaningful</div>...'
```

Further, if this implementation doesn't suit your needs, you can define your own custom highlighter class. As long as it implements the API you've just seen, it can highlight however you choose. For example:

```
# In ``myapp/utils.py``...
from haystack.utils import Highlighter

class BorkHighlighter(Highlighter):
    def render_html(self, highlight_locations=None, start_offset=None, end_offset=None):
        highlighted_chunk = self.text_block[start_offset:end_offset]

        for word in self.query_words:
            highlighted_chunk = highlighted_chunk.replace(word, 'Bork!')

        return highlighted_chunk
```

Then set the `HAYSTACK_CUSTOM_HIGHLIGHTER` setting to `myapp.utils.BorkHighlighter`. Usage would then look like:

```
>>> highlight = BorkHighlighter(my_query)
>>> highlight.highlight(my_text)
u'Bork! that would be more Bork! in real life.'
```

Now the `{% highlight %}` template tag will also use this highlighter.

2.3 Faceting

2.3.1 What Is Faceting?

Faceting is a way to provide users with feedback about the number of documents which match terms they may be interested in. At it's simplest, it gives document counts based on words in the corpus, date ranges, numeric ranges or even advanced queries.

Faceting is particularly useful when trying to provide users with drill-down capabilities. The general workflow in this regard is:

1. You can choose what you want to facet on.
2. The search engine will return the counts it sees for that match.
3. You display those counts to the user and provide them with a link.
4. When the user chooses a link, you narrow the search query to only include those conditions and display the results, potentially with further facets.

Note: Faceting can be difficult, especially in providing the user with the right number of options and/or the right areas to be able to drill into. This is unique to every situation and demands following what real users need.

You may want to consider logging queries and looking at popular terms to help you narrow down how you can help your users.

Haystack provides functionality so that all of the above steps are possible. From the ground up, let's build a faceted search setup. This assumes that you have been to work through the [Getting Started with Haystack](#) and have a working Haystack installation. The same setup from the [Getting Started with Haystack](#) applies here.

2.3.2 1. Determine Facets And SearchQuerySet

Determining what you want to facet on isn't always easy. For our purposes, we'll facet on the `author` field.

In order to facet effectively, the search engine should store both a standard representation of your data as well as exact version to facet on. This is generally accomplished by duplicating the field and storing it via two different types. Duplication is suggested so that those fields are still searchable in the standard ways.

To inform Haystack of this, you simply pass along a `faceted=True` parameter on the field(s) you wish to facet on. So to modify our existing example:

```
class NoteIndex(SearchIndex, indexes.Indexable):
    text = CharField(document=True, use_template=True)
    author = CharField(model_attr='user', faceted=True)
    pub_date = DateTimeField(model_attr='pub_date')
```

Haystack quietly handles all of the backend details for you, creating a similar field to the type you specified with `_exact` appended. Our example would now have both a `author` and `author_exact` field, though this is largely an implementation detail.

To pull faceting information out of the index, we'll use the `SearchQuerySet.facet` method to setup the facet and the `SearchQuerySet.facet_counts` method to retrieve back the counts seen.

Experimenting in a shell (`./manage.py shell`) is a good way to get a feel for what various facets might look like:

```
>>> from haystack.query import SearchQuerySet
>>> sqs = SearchQuerySet().facet('author')
>>> sqs.facet_counts()
{
  'dates': {},
  'fields': {
    'author': [
      ('john', 4),
      ('daniel', 2),
      ('sally', 1),
      ('terry', 1),
    ],
  },
  'queries': {}
}
```

Note: Note that, despite the duplication of fields, you should provide the regular name of the field when faceting. Haystack will intelligently handle the underlying details and mapping.

As you can see, we get back a dictionary which provides access to the three types of facets available: `fields`, `dates` and `queries`. Since we only faceted on the `author` field (which actually facets on the `author_exact` field managed by Haystack), only the `fields` key has any data associated with it. In this case, we have a corpus of eight documents with four unique authors.

Note: Facets are chainable, like most `SearchQuerySet` methods. However, unlike most `SearchQuerySet` methods, they are *NOT* affected by `filter` or similar methods. The only method that has any effect on facets is the `narrow` method (which is how you provide drill-down).

Now that we have the facet we want, it's time to implement it.

2.3.3 2. Switch to the FacetedSearchView and FacetedSearchForm

There are three things that we'll need to do to expose facets to our frontend. The first is construct the `SearchQuerySet` we want to use. We should have that from the previous step. The second is to switch to the `FacetedSearchView`. This view is useful because it prepares the facet counts and provides them in the context as facets.

Optionally, the third step is to switch to the `FacetedSearchForm`. As it currently stands, this is only useful if you want to provide drill-down, though it may provide more functionality in the future. We'll do it for the sake of having it in place but know that it's not required.

In your `URLconf`, you'll need to switch to the `FacetedSearchView`. Your `URLconf` should resemble:

```
from django.conf.urls.defaults import *
from haystack.forms import FacetedSearchForm
from haystack.query import SearchQuerySet
from haystack.views import FacetedSearchView

sqs = SearchQuerySet().facet('author')

urlpatterns = patterns('haystack.views',
    url(r'^$', FacetedSearchView(form_class=FacetedSearchForm, searchqueryset=sqs), name='haystack_s
    )
```

The `FacetedSearchView` will now instantiate the `FacetedSearchForm` and use the `SearchQuerySet` we provided. Now, a `facets` variable will be present in the context. This is added in an overridden `extra_context` method.

2.3.4 3. Display The Facets In The Template

Templating facets involves simply adding an extra bit of processing to display the facets (and optionally to link to provide drill-down). An example template might look like this:

```
<form method="get" action=".">
  <table>
    <tbody>
      <tr>
        <td>&nbsp;</td>
        <td><input type="submit" value="Search"></td>
      </tr>
    </tbody>
  </table>
</form>

{% if query %}
  <!-- Begin faceting. -->
  <h2>By Author</h2>

  <div>
    <dl>
      {% if facets.fields.author %}
        <dt>Author</dt>
        {# Provide only the top 5 authors #}
        {% for author in facets.fields.author|slice:":5" %}
          <dd><a href="{ request.get_full_path }&amp;selected_facets=author_exact:{{ autl
```

```

        {% endfor %}
        {% else %}
        <p>No author facets.</p>
        {% endif %}
    </dl>
</div>
<!-- End faceting -->

<!-- Display results... -->
{% for result in results %}
    <div class="search_result">
        <h3><a href="{{ result.object.get_absolute_url }}">{{ result.object.title }}</a></h3>

        <p>{{ result.object.body|truncatewords:80 }}</p>
    </div>
{% empty %}
    <p>Sorry, no results found.</p>
{% endfor %}
{% endif %}

```

Displaying the facets is a matter of looping through the facets you want and providing the UI to suit. The `author.0` is the facet text from the backend and the `author.1` is the facet count.

2.3.5 4. Narrowing The Search

We've also set ourselves up for the last bit, the drill-down aspect. By appending on the `selected_facets` to the URLs, we're informing the `FacetedSearchForm` that we want to narrow our results to only those containing the author we provided.

For a concrete example, if the facets on author come back as:

```

{
  'dates': {},
  'fields': {
    'author': [
      ('john', 4),
      ('daniel', 2),
      ('sally', 1),
      ('terry', 1),
    ],
  },
  'queries': {}
}

```

You should present a list similar to:

```

<ul>
  <li><a href="/search/?q=Haystack&selected_facets=author_exact:john">john</a> (4)</li>
  <li><a href="/search/?q=Haystack&selected_facets=author_exact:daniel">daniel</a> (2)</li>
  <li><a href="/search/?q=Haystack&selected_facets=author_exact:sally">sally</a> (1)</li>
  <li><a href="/search/?q=Haystack&selected_facets=author_exact:terry">terry</a> (1)</li>
</ul>

```

Warning: Haystack can automatically handle most details around faceting. However, since `selected_facets` is passed directly to `narrow`, it must use the duplicated field name. Improvements to this are planned but incomplete.

This is simply the default behavior but it is possible to override or provide your own form which does additional processing. You could also write your own faceted `SearchView`, which could provide additional/different facets based on facets chosen. There is a wide range of possibilities available to help the user navigate your content.

2.4 Autocomplete

Autocomplete is becoming increasingly common as an add-on to search. Haystack makes it relatively simple to implement. There are two steps in the process, one to prepare the data and one to implement the actual search.

2.4.1 Step 1. Setup The Data

To do autocomplete effectively, the search backend uses n-grams (essentially a small window passed over the string). Because this alters the way your data needs to be stored, the best approach is to add a new field to your `SearchIndex` that contains the text you want to autocomplete on.

You have two choices: `NgramField` & `EdgeNgramField`. Though very similar, the choice of field is somewhat important.

- If you're working with standard text, `EdgeNgramField` tokenizes on whitespace. This prevents incorrect matches when part of two different words are mashed together as one n-gram. **This is what most users should use.**
- If you're working with Asian languages or want to be able to autocomplete across word boundaries, `NgramField` should be what you use.

Example (continuing from the tutorial):

```
import datetime
from haystack import indexes
from myapp.models import Note

class NoteIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    author = indexes.CharField(model_attr='user')
    pub_date = indexes.DateTimeField(model_attr='pub_date')
    # We add this for autocomplete.
    content_auto = indexes.EdgeNgramField(model_attr='content')

    def get_model(self):
        return Note

    def index_queryset(self, using=None):
        """Used when the entire index for model is updated."""
        return Note.objects.filter(pub_date__lte=datetime.datetime.now())
```

As with all schema changes, you'll need to rebuild/update your index after making this change.

2.4.2 Step 2. Performing The Query

Haystack ships with a convenience method to perform most autocomplete searches. You simply provide a field & the query you wish to search on to the `SearchQuerySet.autocomplete` method. Given the previous example, an example search would look like:

```

from haystack.query import SearchQuerySet

SearchQuerySet().autocomplete(content_auto='old')
# Result match things like 'goldfish', 'cuckold' & 'older'.

```

The results from the `SearchQuerySet.autocomplete` method are full search results, just like any regular filter.

If you need more control over your results, you can use standard `SearchQuerySet.filter` calls. For instance:

```

from haystack.query import SearchQuerySet

sqs = SearchQuerySet().filter(content_auto=request.GET.get('q', ''))

```

This can also be extended to use `SQ` for more complex queries (and is what's being done under the hood in the `SearchQuerySet.autocomplete` method).

2.4.3 Example Implementation

The above is the low-level backend portion of how you implement autocomplete. To make it work in browser, you need both a view to run the autocomplete & some Javascript to fetch the results.

Since it comes up often, here is an example implementation of those things.

Warning: This code comes with no warranty. Don't ask for support on it. If you copy-paste it & it burns down your server room, I'm not liable for any of it. It worked this one time on my machine in a simulated environment. And yeah, semicolon-less + 2 space + comma-first. Deal with it.

A stripped-down view might look like:

```

# views.py
import simplejson as json
from django.http import HttpResponse
from haystack.query import SearchQuerySet

def autocomplete(request):
    sqs = SearchQuerySet().autocomplete(request.GET.get('q', ''))[:5]
    suggestions = [result.title for result in sqs]
    # Make sure you return a JSON object, not a bare list.
    # Otherwise, you could be vulnerable to an XSS attack.
    the_data = json.dumps({
        'results': suggestions
    })
    return HttpResponse(the_data, content_type='application/json')

```

The template might look like:

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Autocomplete Example</title>
</head>
<body>
  <h1>Autocomplete Example</h1>

```

```
<form method="post" action="/search/" class="autocomplete-me">
  <input type="text" id="id_q" name="q">
  <input type="submit" value="Search!">
</form>

<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.8.3/jquery.min.js"></script>
<script type="text/javascript">
  // In a perfect world, this would be its own library file that got included
  // on the page & only the ``$(document).ready(...)`` below would be present.
  // But this is an example.
  var Autocomplete = function(options) {
    this.form_selector = options.form_selector
    this.url = options.url || '/search/autocomplete/'
    this.delay = parseInt(options.delay || 300)
    this.minimum_length = parseInt(options.minimum_length || 3)
    this.form_elem = null
    this.query_box = null
  }

  Autocomplete.prototype.setup = function() {
    var self = this

    this.form_elem = $(this.form_selector)
    this.query_box = this.form_elem.find('input[name=q]')

    // Watch the input box.
    this.query_box.on('keyup', function() {
      var query = self.query_box.val()

      if(query.length < self.minimum_length) {
        return false
      }

      self.fetch(query)
    })

    // On selecting a result, populate the search field.
    this.form_elem.on('click', '.ac-result', function(ev) {
      self.query_box.val($(this).text())
      $('.ac-results').remove()
      return false
    })
  }

  Autocomplete.prototype.fetch = function(query) {
    var self = this

    $.ajax({
      url: this.url
    , data: {
        'q': query
      }
    , success: function(data) {
        self.show_results(data)
      }
    })
  }
}
```

```

Autocomplete.prototype.show_results = function(data) {
  // Remove any existing results.
  $('.ac-results').remove()

  var results = data.results || []
  var results_wrapper = $('<div class="ac-results"></div>')
  var base_elem = $('<div class="result-wrapper"><a href="#" class="ac-result"></a></div>')

  if(results.length > 0) {
    for(var res_offset in results) {
      var elem = base_elem.clone()
      // Don't use .html(...) here, as you open yourself to XSS.
      // Really, you should use some form of templating.
      elem.find('.ac-result').text(results[res_offset])
      results_wrapper.append(elem)
    }
  }
  else {
    var elem = base_elem.clone()
    elem.text("No results found.")
    results_wrapper.append(elem)
  }

  this.query_box.after(results_wrapper)
}

$(document).ready(function() {
  window.autocomplete = new Autocomplete({
    form_selector: '.autocomplete-me'
  })
  window.autocomplete.setup()
})
</script>
</body>
</html>

```

2.5 Boost

Scoring is a critical component of good search. Normal full-text searches automatically score a document based on how well it matches the query provided. However, sometimes you want certain documents to score better than they otherwise would. Boosting is a way to achieve this. There are three types of boost:

- Term Boost
- Document Boost
- Field Boost

Note: Document & Field boost support was added in Haystack 1.1.

Despite all being types of boost, they take place at different times and have slightly different effects on scoring.

Term boost happens at query time (when the search query is run) and is based around increasing the score if a certain word/phrase is seen.

On the other hand, document & field boosts take place at indexing time (when the document is being added to the index). Document boost causes the relevance of the entire result to go up, where field boost causes only searches within that field to do better.

Warning: Be warned that boost is very, very sensitive & can hurt overall search quality if over-zealously applied. Even very small adjustments can affect relevance in a big way.

2.5.1 Term Boost

Term boosting is achieved by using `SearchQuerySet.boost`. You provide it the term you want to boost on & a floating point value (based around 1.0 as 100% - no boost).

Example:

```
# Slight increase in relevance for documents that include "banana".
sqs = SearchQuerySet().boost('banana', 1.1)

# Big decrease in relevance for documents that include "blueberry".
sqs = SearchQuerySet().boost('blueberry', 0.8)
```

See the [SearchQuerySet API docs](#) for more details on using this method.

2.5.2 Document Boost

Document boosting is done by adding a `boost` field to the prepared data `SearchIndex` creates. The best way to do this is to override `SearchIndex.prepare`:

```
from haystack import indexes
from notes.models import Note

class NoteSearchIndex(indexes.SearchIndex, indexes.Indexable):
    # Your regular fields here then...

    def prepare(self, obj):
        data = super(NoteSearchIndex, self).prepare(obj)
        data['boost'] = 1.1
        return data
```

Another approach might be to add a new field called `boost`. However, this can skew your schema and is not encouraged.

2.5.3 Field Boost

Field boosting is enabled by setting the `boost` kwarg on the desired field. An example of this might be increasing the significance of a `title`:

```
from haystack import indexes
from notes.models import Note

class NoteSearchIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    title = indexes.CharField(model_attr='title', boost=1.125)
```



```
def get_model(self):
    return Note
```

2.6 Signal Processors

Keeping data in sync between the (authoritative) database & the (non-authoritative) search index is one of the more difficult problems when using Haystack. Even frequently running the `update_index` management command still introduces lag between when the data is stored & when it's available for searching.

A solution to this is to incorporate Django's signals (specifically `models.db.signals.post_save` & `models.db.signals.post_delete`), which then trigger *individual* updates to the search index, keeping them in near-perfect sync.

Older versions of Haystack (pre-v2.0) tied the `SearchIndex` directly to the signals, which caused occasional conflicts of interest with third-party applications.

To solve this, starting with Haystack v2.0, the concept of a `SignalProcessor` has been introduced. In its simplest form, the `SignalProcessor` listens to whatever signals are setup & can be configured to then trigger the updates without having to change any `SearchIndex` code.

Warning: Incorporating Haystack's `SignalProcessor` into your setup **will** increase the overall load (CPU & perhaps I/O depending on configuration). You will need to capacity plan for this & ensure you can make the tradeoff of more real-time results for increased load.

2.6.1 Default - `BaseSignalProcessor`

The default setup is configured to use the `haystack.signals.BaseSignalProcessor` class, which includes all the underlying code necessary to handle individual updates/deletes, **BUT DOES NOT HOOK UP THE SIGNALS**.

This means that, by default, **NO ACTION IS TAKEN BY HAYSTACK** when a model is saved or deleted. The `BaseSignalProcessor.setup` & `BaseSignalProcessor.teardown` methods are both empty to prevent anything from being setup at initialization time.

This usage is configured very simply (again, by default) with the `HAYSTACK_SIGNAL_PROCESSOR` setting. An example of manually setting this would look like:

```
HAYSTACK_SIGNAL_PROCESSOR = 'haystack.signals.BaseSignalProcessor'
```

This class forms an excellent base if you'd like to override/extend for more advanced behavior. Which leads us to...

2.6.2 Realtime - `RealtimeSignalProcessor`

The other included `SignalProcessor` is the `haystack.signals.RealtimeSignalProcessor` class. It is an extremely thin extension of the `BaseSignalProcessor` class, differing only in that it implements the `setup/teardown` methods, tying **ANY** Model save/delete to the signal processor.

If the model has an associated `SearchIndex`, the `RealtimeSignalProcessor` will then trigger an update/delete of that model instance within the search index proper.

Configuration looks like:

```
HAYSTACK_SIGNAL_PROCESSOR = 'haystack.signals.RealtimeSignalProcessor'
```

This causes **all** `SearchIndex` classes to work in a realtime fashion.

Note: These updates happen in-process, which if a request-response cycle is involved, may cause the user with the browser to sit & wait for indexing to be completed. Since this wait can be undesirable, especially under load, you may wish to look into queued search options. See the *Haystack-Related Applications* documentation for existing options.

2.6.3 Custom SignalProcessors

The `BaseSignalProcessor` & `RealtimeSignalProcessor` classes are fairly simple/straightforward to customize or extend. Rather than forking Haystack to implement your modifications, you should create your own subclass within your codebase (anywhere that's importable is usually fine, though you should avoid `models.py` files).

For instance, if you only wanted `User` saves to be realtime, deferring all other updates to the management commands, you'd implement the following code:

```
from django.contrib.auth.models import User
from django.db import models
from haystack import signals

class UserOnlySignalProcessor(BaseSignalProcessor):
    def setup(self):
        # Listen only to the ``User`` model.
        models.signals.post_save.connect(self.handle_save, sender=User)
        models.signals.post_delete.connect(self.handle_delete, sender=User)

    def teardown(self):
        # Disconnect only for the ``User`` model.
        models.signals.post_save.disconnect(self.handle_save, sender=User)
        models.signals.post_delete.disconnect(self.handle_delete, sender=User)
```

For other customizations (modifying how saves/deletes should work), you'll need to override/extend the `handle_save`/`handle_delete` methods. The source code is your best option for referring to how things currently work on your version of Haystack.

2.7 Multiple Indexes

Much like Django's [multiple database support](#), Haystack has “multiple index” support. This allows you to talk to several different engines at the same time. It enables things like master-slave setups, multiple language indexing, separate indexes for general search & autocomplete as well as other options.

2.7.1 Specifying Available Connections

You can supply as many backends as you like, each with a descriptive name. A complete setup that accesses all backends might look like:

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.solr_backend.SolrEngine',
        'URL': 'http://localhost:9001/solr/default',
```

```

        'TIMEOUT': 60 * 5,
        'INCLUDE_SPELLING': True,
        'BATCH_SIZE': 100,
        'SILENTLY_FAIL': True,
    },
    'autocomplete': {
        'ENGINE': 'haystack.backends.whoosh_backend.WhooshEngine',
        'PATH': '/home/search/whoosh_index',
        'STORAGE': 'file',
        'POST_LIMIT': 128 * 1024 * 1024,
        'INCLUDE_SPELLING': True,
        'BATCH_SIZE': 100,
        'SILENTLY_FAIL': True,
    },
    'slave': {
        'ENGINE': 'xapian_backend.XapianEngine',
        'PATH': '/home/search/xapian_index',
        'INCLUDE_SPELLING': True,
        'BATCH_SIZE': 100,
        'SILENTLY_FAIL': True,
    },
    'db': {
        'ENGINE': 'haystack.backends.simple_backend.SimpleEngine',
        'SILENTLY_FAIL': True,
    }
}

```

You are required to have at least one connection listed within `HAYSTACK_CONNECTIONS`, it must be named `default` & it must have a valid `ENGINE` within it.

2.7.2 Management Commands

All management commands that manipulate data use **ONLY** one connection at a time. By default, they use the `default` index but accept a `--using` flag to specify a different connection. For example:

```
./manage.py rebuild_index --noinput --using=whoosh
```

2.7.3 Automatic Routing

To make the selection of the correct index easier, Haystack (like Django) has the concept of “routers”. All provided routers are checked whenever a read or write happens, stopping at the first router that knows how to handle it.

Haystack ships with a `DefaultRouter` enabled. It looks like:

```

class DefaultRouter(BaseRouter):
    def for_read(self, **hints):
        return DEFAULT_ALIAS

    def for_write(self, **hints):
        return DEFAULT_ALIAS

```

On a read (when a search query is executed), the `DefaultRouter.for_read` method is checked & returns the `DEFAULT_ALIAS` (which is `default`), telling whatever requested it that it should perform the query against the `default` connection. The same process is followed for writes.

If the `for_read` or `for_write` method returns `None`, that indicates that the current router can’t handle the data. The next router is then checked.

The `hints` passed can be anything that helps the router make a decision. This data should always be considered optional & be guarded against. At current, `for_write` receives an `index` option (pointing to the `SearchIndex` calling it) while `for_read` may receive `models` (being a list of `Model` classes the `SearchQuerySet` may be looking at).

You may provide as many routers as you like by overriding the `HAYSTACK_ROUTERS` setting. For example:

```
HAYSTACK_ROUTERS = ['myapp.routers.MasterRouter', 'myapp.routers.SlaveRouter', 'haystack.routers.DefaultRouter']
```

Master-Slave Example

The `MasterRouter` & `SlaveRouter` might look like:

```
from haystack import routers

class MasterRouter(routers.BaseRouter):
    def for_write(self, **hints):
        return 'master'

    def for_read(self, **hints):
        return None

class SlaveRouter(routers.BaseRouter):
    def for_write(self, **hints):
        return None

    def for_read(self, **hints):
        return 'slave'
```

The observant might notice that since the methods don't overlap, this could be combined into one `Router` like so:

```
from haystack import routers

class MasterSlaveRouter(routers.BaseRouter):
    def for_write(self, **hints):
        return 'master'

    def for_read(self, **hints):
        return 'slave'
```

2.7.4 Manually Selecting

There may be times when automatic selection of the correct index is undesirable, such as when fixing erroneous data in an index or when you know exactly where data should be located.

For this, the `SearchQuerySet` class allows for manually selecting the index via the `SearchQuerySet.using` method:

```
from haystack.query import SearchQuerySet

# Uses the routers' opinion.
sqs = SearchQuerySet().auto_query('banana')

# Forces the default.
```

```
sqs = SearchQuerySet().using('default').auto_query('banana')

# Forces the slave connection (presuming it was setup).
sqs = SearchQuerySet().using('slave').auto_query('banana')
```

Warning: Note that the models a `SearchQuerySet` is trying to pull from must all come from the same index. Haystack is not able to combine search queries against different indexes.

2.7.5 Custom Index Selection

If a specific backend has been selected, the `SearchIndex.index_queryset` and `SearchIndex.read_queryset` will receive the backend name, giving indexes the opportunity to customize the returned queryset.

For example, a site which uses separate indexes for recent items and older content might define `index_queryset` to filter the items based on date:

```
def index_queryset(self, using=None):
    qs = Note.objects.all()
    archive_limit = datetime.datetime.now() - datetime.timedelta(days=90)

    if using == "archive":
        return qs.filter(pub_date__lte=archive_limit)
    else:
        return qs.filter(pub_date__gte=archive_limit)
```

Multi-lingual Content

Most search engines require you to set the language at the index level. For example, a multi-lingual site using Solr can use [multiple cores](#) and corresponding Haystack backends using the language name. Under this scenario, queries are simple:

```
sqs = SearchQuerySet.using(lang).auto_query(...)
```

During index updates, the Index's `index_queryset` method will need to filter the items to avoid sending the wrong content to the search engine:

```
def index_queryset(self, using=None):
    return Post.objects.filter(language=using)
```

2.8 Rich Content Extraction

For some projects it is desirable to index text content which is stored in structured files such as PDFs, Microsoft Office documents, images, etc. Currently only Solr's `ExtractingRequestHandler` is directly supported by Haystack but the approach below could be used with any backend which supports this feature.

2.8.1 Extracting Content

`SearchBackend.extract_file_contents()` accepts a file or file-like object and returns a dictionary containing two keys: `metadata` and `contents`. The `contents` value will be a string containing all of the text which

the backend managed to extract from the file contents. `metadata` will always be a dictionary but the keys and values will vary based on the underlying extraction engine and the type of file provided.

2.8.2 Indexing Extracted Content

Generally you will want to include the extracted text in your main document field along with everything else specified in your search template. This example shows how to override a hypothetical `FileIndex`'s `prepare` method to include the extract content along with information retrieved from the database:

```
def prepare(self, obj):
    data = super(FileIndex, self).prepare(obj)

    # This could also be a regular Python open() call, a StringIO instance
    # or the result of opening a URL. Note that due to a library limitation
    # file_obj must have a .name attribute even if you need to set one
    # manually before calling extract_file_contents:
    file_obj = obj.the_file.open()

    extracted_data = self.backend.extract_file_contents(file_obj)

    # Now we'll finally perform the template processing to render the
    # text field with *all* of our metadata visible for templating:
    t = loader.select_template(('search/indexes/myapp/file_text.txt', ))
    data['text'] = t.render(Context({'object': obj,
                                   'extracted': extracted_data}))

    return data
```

This allows you to insert the extracted text at the appropriate place in your template, modified or intermixed with database content as appropriate:

```
{{ object.title }}
{{ object.owner.name }}

...

{% for k, v in extracted.metadata.items %}
    {% for val in v %}
        {{ k }}: {{ val|safe }}
    {% endfor %}
{% endfor %}

{{ extracted.contents|striptags|safe }}
```

2.9 Spatial Search

Spatial search (also called geospatial search) allows you to take data that has a geographic location & enhance the search results by limiting them to a physical area. Haystack, combined with the latest versions of a couple engines, can provide this type of search.

In addition, Haystack tries to implement these features in a way that is as close to [GeoDjango](#) as possible. There are some differences, which we'll highlight throughout this guide. Additionally, while the support isn't as comprehensive as PostGIS (for example), it is still quite useful.

2.9.1 Additional Requirements

The spatial functionality has only one non-included, non-available-in-Django dependency:

- `geopy - pip install geopy`

If you do not ever need distance information, you may be able to skip installing `geopy`.

2.9.2 Support

You need the latest & greatest of either Solr or Elasticsearch. None of the other backends (specifially the engines) support this kind of search.

For `Solr`, you'll need at least **v3.5+**. In addition, if you have an existing install of Haystack & Solr, you'll need to upgrade the schema & reindex your data. If you're adding geospatial data, you would have to reindex anyhow.

For `Elasticsearch`, you'll need at least v0.17.7, preferably v0.18.6 or better. If you're adding geospatial data, you'll have to reindex as well.

Lookup Type	Solr	Elasticsearch	Whoosh	Xapian	Simple
<i>within</i>	X	X			
<i>dwithin</i>	X	X			
<i>distance</i>	X	X			
<i>order_by('distance')</i>	X	X			
<i>polygon</i>		X			

For more details, you can inspect <http://wiki.apache.org/solr/SpatialSearch> or <http://www.elasticsearch.org/guide/reference/query-dsl/geo-bounding-box-filter.html>.

2.9.3 Geospatial Assumptions

Points

Haystack prefers to work with `Point` objects, which are located in `django.contrib.gis.geos.Point` but conveniently importable out of `haystack.utils.geo.Point`.

`Point` objects use **LONGITUDE, LATITUDE** for their construction, regardless if you use the parameters to instantiate them or `WKT/GEOSGeometry`.

Examples:

```
# Using positional arguments.
from haystack.utils.geo import Point
pnt = Point(-95.23592948913574, 38.97127105172941)

# Using WKT.
from django.contrib.gis.geos import GEOSGeometry
pnt = GEOSGeometry('POINT(-95.23592948913574 38.97127105172941)')
```

They are preferred over just providing `latitude, longitude` because they are more intelligent, have a spatial reference system attached & are more consistent with `GeoDjango`'s use.

Distance

Haystack also uses the `D` (or `Distance`) objects from `GeoDjango`, implemented in `django.contrib.gis.measure.Distance` but conveniently importable out of `haystack.utils.geo.D` (or `haystack.utils.geo.Distance`).

Distance objects accept a very flexible set of measurements during instantiation and can convert amongst them freely. This is important, because the engines rely on measurements being in kilometers but you're free to use whatever units you want.

Examples:

```
from haystack.utils.geo import D

# Start at 5 miles.
imperial_d = D(mi=5)

# Convert to fathoms...
fathom_d = imperial_d.fathom

# Now to kilometers...
km_d = imperial_d.km

# And back to miles.
mi = imperial_d.mi
```

They are preferred over just providing a raw distance because they are more intelligent, have a well-defined unit system attached & are consistent with GeoDjango's use.

WGS-84

All engines assume WGS-84 (SRID 4326). At the time of writing, there does **not** appear to be a way to switch this. Haystack will transform all points into this coordinate system for you.

2.9.4 Indexing

Indexing is relatively simple. Simply add a `LocationField` (or several) onto your `SearchIndex` class(es) & provide them a `Point` object. For example:

```
from haystack import indexes
from shops.models import Shop

class ShopIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    # ... the usual, then...
    location = indexes.LocationField(model_attr='coordinates')

    def get_model(self):
        return Shop
```

If you must manually prepare the data, you have to do something slightly less convenient, returning a string-ified version of the coordinates in WGS-84 as `lat, long`:

```
from haystack import indexes
from shops.models import Shop

class ShopIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    # ... the usual, then...
    location = indexes.LocationField()
```



```

def get_model(self):
    return Shop

def prepare_location(self, obj):
    # If you're just storing the floats...
    return "%s,%s" % (obj.latitude, obj.longitude)

```

Alternatively, you could build a method/property onto the Shop model that returns a Point based on those coordinates:

```

# shops/models.py
from django.contrib.gis.geos import Point
from django.db import models

class Shop(models.Model):
    # ... the usual, then...
    latitude = models.FloatField()
    longitude = models.FloatField()

    # Usual methods, then...
    def get_location(self):
        # Remember, longitude FIRST!
        return Point(self.longitude, self.latitude)

# shops/search_indexes.py
from haystack import indexes
from shops.models import Shop

class ShopIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    location = indexes.LocationField(model_attr='get_location')

    def get_model(self):
        return Shop

```

2.9.5 Querying

There are two types of geospatial queries you can run, `within` & `dwithin`. Like their GeoDjango counterparts (`within` & `dwithin`), these methods focus on finding results within an area.

`within`

`SearchQuerySet.within(self, field, point_1, point_2)`

`within` is a bounding box comparison. A bounding box is a rectangular area within which to search. It's composed of a bottom-left point & a top-right point, though provided you give two opposing corners in either order, Haystack will determine the right coordinates. It is faster but slightly sloppier than its counterpart.

Examples:

```

from haystack.query import SearchQuerySet
from haystack.utils.geo import Point

```

```
downtown_bottom_left = Point(-95.23947, 38.9637903)
downtown_top_right = Point(-95.23362278938293, 38.973081081164715)

# 'location' is the fieldname from our ``SearchIndex``...

# Do the bounding box query.
sqs = SearchQuerySet().within('location', downtown_bottom_left, downtown_top_right)

# Can be chained with other Haystack calls.
sqs = SearchQuerySet().auto_query('coffee').within('location', downtown_bottom_left, downtown_top_right)
```

Note: In GeoDjango, assuming the Shop model had been properly geo-ified, this would have been implemented as:

```
from shops.models import Shop
Shop.objects.filter(location__within=(downtown_bottom_left, downtown_top_right))
```

Haystack's form differs because it yielded a cleaner implementation, was no more typing than the GeoDjango version & tried to maintain the same terminology/similar signature.

dwwithin

SearchQuerySet.**dwwithin**(*self, field, point, distance*)

dwwithin is a radius-based search. A radius-based search is a circular area within which to search. It's composed of a center point & a radius (in kilometers, though Haystack will use the D object's conversion utilities to get it there). It is slower than "within" but very exact & can involve fewer calculations on your part.

Examples:

```
from haystack.query import SearchQuerySet
from haystack.utils.geo import Point, D

ninth_and_mass = Point(-95.23592948913574, 38.96753407043678)
# Within a two miles.
max_dist = D(mi=2)

# 'location' is the fieldname from our ``SearchIndex``...

# Do the radius query.
sqs = SearchQuerySet().dwwithin('location', ninth_and_mass, max_dist)

# Can be chained with other Haystack calls.
sqs = SearchQuerySet().auto_query('coffee').dwwithin('location', ninth_and_mass, max_dist).order_by('')
```

Note: In GeoDjango, assuming the Shop model had been properly geo-ified, this would have been implemented as:

```
from shops.models import Shop
Shop.objects.filter(location__dwwithin=(ninth_and_mass, D(mi=2)))
```

Haystack's form differs because it yielded a cleaner implementation, was no more typing than the GeoDjango version & tried to maintain the same terminology/similar signature.

distance

`SearchQuerySet.distance(self, field, point)`

By default, search results will come back without distance information attached to them. In the concept of a bounding box, it would be ambiguous what the distances would be calculated against. And it is more calculation that may not be necessary.

So like GeoDjango, Haystack exposes a method to signify that you want to include these calculated distances on results.

Examples:

```

from haystack.query import SearchQuerySet
from haystack.utils.geo import Point, D

ninth_and_mass = Point(-95.23592948913574, 38.96753407043678)

# On a bounding box...
downtown_bottom_left = Point(-95.23947, 38.9637903)
downtown_top_right = Point(-95.23362278938293, 38.973081081164715)

sqs = SearchQuerySet().within('location', downtown_bottom_left, downtown_top_right).distance('location', ninth_and_mass)

# ...Or on a radius query.
sqs = SearchQuerySet().dwithin('location', ninth_and_mass, D(mi=2)).distance('location', ninth_and_mass)

```

You can even apply a different field, for instance if you calculate results of key, well-cached hotspots in town but want distances from the user's current position:

```

from haystack.query import SearchQuerySet
from haystack.utils.geo import Point, D

ninth_and_mass = Point(-95.23592948913574, 38.96753407043678)
user_loc = Point(-95.23455619812012, 38.97240128290697)

sqs = SearchQuerySet().dwithin('location', ninth_and_mass, D(mi=2)).distance('location', user_loc)

```

Note: The astute will notice this is Haystack's biggest departure from GeoDjango. In GeoDjango, this would have been implemented as:

```

from shops.models import Shop
Shop.objects.filter(location__dwithin=(ninth_and_mass, D(mi=2))).distance(user_loc)

```

Note that, by default, the GeoDjango form leaves *out* the field to be calculating against (though it's possible to override it & specify the field).

Haystack's form differs because the same assumptions are difficult to make. GeoDjango deals with a single model at a time, where Haystack deals with a broad mix of models. Additionally, accessing `Model` information is a couple hops away, so Haystack favors the explicit (if slightly more typing) approach.

2.9.6 Ordering

Because you're dealing with search, even with geospatial queries, results still come back in **RELEVANCE** order. If you want to offer the user ordering results by distance, there's a simple way to enable this ordering.

Using the standard Haystack `order_by` method, if you specify `distance` or `-distance` **ONLY**, you'll get geographic ordering. Additionally, you must have a call to `.distance()` somewhere in the chain, otherwise there is no distance information on the results & nothing to sort by.

Examples:

```
from haystack.query import SearchQuerySet
from haystack.utils.geo import Point, D

ninth_and_mass = Point(-95.23592948913574, 38.96753407043678)
downtown_bottom_left = Point(-95.23947, 38.9637903)
downtown_top_right = Point(-95.23362278938293, 38.973081081164715)

# Non-geo ordering.
sqs = SearchQuerySet().within('location', downtown_bottom_left, downtown_top_right).order_by('title')
sqs = SearchQuerySet().within('location', downtown_bottom_left, downtown_top_right).distance('location')

# Geo ordering, closest to farthest.
sqs = SearchQuerySet().within('location', downtown_bottom_left, downtown_top_right).distance('location')
# Geo ordering, farthest to closest.
sqs = SearchQuerySet().dwithin('location', ninth_and_mass, D(mi2)).distance('location', ninth_and_ma
```

Note: This call is identical to the GeoDjango usage.

Warning: You can not specify both a distance & lexicographic ordering. If you specify more than just distance or `-distance`, Haystack assumes distance is a field in the index & tries to sort on it. Example:

```
# May blow up!
sqs = SearchQuerySet().dwithin('location', ninth_and_mass, D(mi2)).distance('location', ninth_and_ma
```

This is a limitation in the engine's implementation.

If you actually **have** a field called `distance` (& aren't using calculated distance information), Haystack will do the right thing in these circumstances.

2.9.7 Caveats

In all cases, you may call the `within/dwithin/distance` methods as many times as you like. However, the **LAST** call is the information that will be used. No combination logic is available, as this is largely a backend limitation.

Combining calls to both `within` & `dwithin` may yield unexpected or broken results. They don't overlap when performing queries, so it may be possible to construct queries that work. Your Mileage May Vary.

If you're an experienced user and are looking for a reference, you may be looking for API documentation and advanced usage as detailed in:

3.1 SearchQuerySet API

`class SearchQuerySet (using=None, query=None)`

The `SearchQuerySet` class is designed to make performing a search and iterating over its results easy and consistent. For those familiar with Django's ORM `QuerySet`, much of the `SearchQuerySet` API should feel familiar.

3.1.1 Why Follow QuerySet?

A couple reasons to follow (at least in part) the `QuerySet` API:

1. Consistency with Django
2. Most Django programmers have experience with the ORM and can use this knowledge with `SearchQuerySet`.

And from a high-level perspective, `QuerySet` and `SearchQuerySet` do very similar things: given certain criteria, provide a set of results. Both are powered by multiple backends, both are abstractions on top of the way a query is performed.

3.1.2 Quick Start

For the impatient:

```
from haystack.query import SearchQuerySet
all_results = SearchQuerySet().all()
hello_results = SearchQuerySet().filter(content='hello')
hello_world_results = SearchQuerySet().filter(content='hello world')
unfriendly_results = SearchQuerySet().exclude(content='hello').filter(content='world')
recent_results = SearchQuerySet().order_by('-pub_date')[:5]

# Using the new input types...
from haystack.inputs import AutoQuery, Exact, Clean
sq = SearchQuerySet().filter(content=AutoQuery(request.GET['q']), product_type=Exact('ancient book'))
```

```
if request.GET['product_url']:
    sqs = sqs.filter(product_url=Clean(request.GET['product_url']))
```

For more on the `AutoQuery`, `Exact`, `Clean` classes & friends, see the *Input Types* documentation.

3.1.3 SearchQuerySet

By default, `SearchQuerySet` provide the documented functionality. You can extend with your own behavior by simply subclassing from `SearchQuerySet` and adding what you need, then using your subclass in place of `SearchQuerySet`.

Most methods in `SearchQuerySet` “chain” in a similar fashion to `QuerySet`. Additionally, like `QuerySet`, `SearchQuerySet` is lazy (meaning it evaluates the query as late as possible). So the following is valid:

```
from haystack.query import SearchQuerySet
results = SearchQuerySet().exclude(content='hello').filter(content='world').order_by('-pub_date').bo
```

3.1.4 The content Shortcut

Searching your document fields is a very common activity. To help mitigate possible differences in `SearchField` names (and to help the backends deal with search queries that inspect the main corpus), there is a special field called `content`. You may use this in any place that other fields names would work (e.g. `filter`, `exclude`, etc.) to indicate you simply want to search the main documents.

For example:

```
from haystack.query import SearchQuerySet

# This searches whatever fields were marked ``document=True``.
results = SearchQuerySet().exclude(content='hello')
```

This special pseudo-field works best with the `exact` lookup and may yield strange or unexpected results with the other lookups.

3.1.5 SearchQuerySet Methods

The primary interface to search in Haystack is through the `SearchQuerySet` object. It provides a clean, programmatic, portable API to the search backend. Many aspects are also “chainable”, meaning you can call methods one after another, each applying their changes to the previous `SearchQuerySet` and further narrowing the search.

All `SearchQuerySet` objects implement a list-like interface, meaning you can perform actions like getting the length of the results, accessing a result at an offset or even slicing the result list.

Methods That Return A SearchQuerySet

`all`

`SearchQuerySet.all(self) :`

Returns all results for the query. This is largely a no-op (returns an identical copy) but useful for denoting exactly what behavior is going on.

`none`

`SearchQuerySet.none(self)` :

Returns an `EmptySearchQuerySet` that behaves like a `SearchQuerySet` but always yields no results.

`filter`

`SearchQuerySet.filter(self, **kwargs)`

Filters the search by looking for (and including) certain attributes.

The lookup parameters (`**kwargs`) should follow the *Field lookups* below. If you specify more than one pair, they will be joined in the query according to the `HAYSTACK_DEFAULT_OPERATOR` setting (defaults to `AND`).

You can pass it either strings or a variety of *Input Types* if you need more advanced query behavior.

Warning: Any data you pass to `filter` gets auto-escaped. If you need to send non-escaped data, use the `Raw` input type (*Input Types*).

Also, if a string with one or more spaces in it is specified as the value, the string will get passed along **AS IS**. This will mean that it will **NOT** be treated as a phrase (like Haystack 1.X's behavior).

If you want to match a phrase, you should use either the `__exact` filter type or the `Exact` input type (*Input Types*).

Examples:

```
sqs = SearchQuerySet().filter(content='foo')

sqs = SearchQuerySet().filter(content='foo', pub_date__lte=datetime.date(2008, 1, 1))

# Identical to the previous example.
sqs = SearchQuerySet().filter(content='foo').filter(pub_date__lte=datetime.date(2008, 1, 1))

# To send unescaped data:
from haystack.inputs import Raw
sqs = SearchQuerySet().filter(title=Raw(trusted_query))

# To use auto-query behavior on a non-`document=True` field.
from haystack.inputs import AutoQuery
sqs = SearchQuerySet().filter(title=AutoQuery(user_query))
```

`exclude`

`SearchQuerySet.exclude(self, **kwargs)`

Narrows the search by ensuring certain attributes are not included.

Warning: Any data you pass to `exclude` gets auto-escaped. If you need to send non-escaped data, use the `Raw` input type (*Input Types*).

Example:

```
sqs = SearchQuerySet().exclude(content='foo')
```

`filter_and`

`SearchQuerySet.filter_and(self, **kwargs)`

Narrows the search by looking for (and including) certain attributes. Join behavior in the query is forced to be AND. Used primarily by the `filter` method.

`filter_or`

`SearchQuerySet.filter_or(self, **kwargs)`

Narrows the search by looking for (and including) certain attributes. Join behavior in the query is forced to be OR. Used primarily by the `filter` method.

`order_by`

`SearchQuerySet.order_by(self, *args)`

Alters the order in which the results should appear. Arguments should be strings that map to the attributes/fields within the index. You may specify multiple fields by comma separating them:

```
SearchQuerySet().filter(content='foo').order_by('author', 'pub_date')
```

Default behavior is ascending order. To specify descending order, prepend the string with a `-`:

```
SearchQuerySet().filter(content='foo').order_by('-pub_date')
```

Note: In general, ordering is locale-specific. Haystack makes no effort to try to reconcile differences between characters from different languages. This means that accented characters will sort closely with the same character and **NOT** necessarily close to the unaccented form of the character.

If you want this kind of behavior, you should override the `prepare_FOO` methods on your `SearchIndex` objects to transliterate the characters as you see fit.

`highlight`

`SearchQuerySet.highlight(self)`

If supported by the backend, the `SearchResult` objects returned will include a highlighted version of the result:

```
sqs = SearchQuerySet().filter(content='foo').highlight()
result = sqs[0]
result.highlighted['text'][0] # u'Two computer scientists walk into a bar. The bartender says "<em>F
```

`models`

`SearchQuerySet.models(self, *models)`

Accepts an arbitrary number of `Model` classes to include in the search. This will narrow the search results to only include results from the models specified.

Example:


```
SearchQuerySet().filter(content='foo').models(BlogEntry, Comment)
```

result_class

SearchQuerySet.**result_class** (*self, klass*)

Allows specifying a different class to use for results.

Overrides any previous usages. If None is provided, Haystack will revert back to the default SearchResult object.

Example:

```
SearchQuerySet().result_class(CustomResult)
```

boost

SearchQuerySet.**boost** (*self, term, boost_value*)

Boosts a certain term of the query. You provide the term to be boosted and the value is the amount to boost it by. Boost amounts may be either an integer or a float.

Example:

```
SearchQuerySet().filter(content='foo').boost('bar', 1.5)
```

facet

SearchQuerySet.**facet** (*self, field, **options*)

Adds faceting to a query for the provided field. You provide the field (from one of the SearchIndex classes) you like to facet on. Any keyword options you provide will be passed along to the backend for that facet.

Example:

```
# For SOLR (setting f.author.facet.*; see http://wiki.apache.org/solr/SimpleFacetParameters#Parameters)
SearchQuerySet().facet('author', mincount=1, limit=10)
# For Elasticsearch (see http://www.elasticsearch.org/guide/reference/api/search/facets/terms-facet.html)
SearchQuerySet().facet('author', size=10, order='term')
```

In the search results you get back, facet counts will be populated in the SearchResult object. You can access them via the facet_counts method.

Example:

```
# Count document hits for each author within the index.
SearchQuerySet().filter(content='foo').facet('author')
```

date_facet

SearchQuerySet.**date_facet** (*self, field, start_date, end_date, gap_by, gap_amount=1*)

Adds faceting to a query for the provided field by date. You provide the field (from one of the SearchIndex classes) you like to facet on, a start_date (either datetime.datetime or datetime.date), an end_date and the amount of time between gaps as gap_by (one of 'year', 'month', 'day', 'hour', 'minute' or 'second').

You can also optionally provide a `gap_amount` to specify a different increment than 1. For example, specifying gaps by week (every seven days) would be `gap_by='day', gap_amount=7`).

In the search results you get back, facet counts will be populated in the `SearchResult` object. You can access them via the `facet_counts` method.

Example:

```
# Count document hits for each day between 2009-06-07 to 2009-07-07 within the index.
SearchQuerySet().filter(content='foo').date_facet('pub_date', start_date=datetime.date(2009, 6, 7), end_date=datetime.date(2009, 7, 7))
```

`query_facet`

`SearchQuerySet.query_facet(self, field, query)`

Adds faceting to a query for the provided field with a custom query. You provide the field (from one of the `SearchIndex` classes) you like to facet on and the backend-specific query (as a string) you'd like to execute.

Please note that this is **NOT** portable between backends. The syntax is entirely dependent on the backend. No validation/cleansing is performed and it is up to the developer to ensure the query's syntax is correct.

In the search results you get back, facet counts will be populated in the `SearchResult` object. You can access them via the `facet_counts` method.

Example:

```
# Count document hits for authors that start with 'jo' within the index.
SearchQuerySet().filter(content='foo').query_facet('author', 'jo*')
```

`within`

`SearchQuerySet.within(self, field, point_1, point_2):`

Spatial: Adds a bounding box search to the query.

See the *Spatial Search* docs for more information.

`dwithin`

`SearchQuerySet.dwithin(self, field, point, distance):`

Spatial: Adds a distance-based search to the query.

See the *Spatial Search* docs for more information.

`stats`

`SearchQuerySet.stats(self, field):`

Adds stats to a query for the provided field. This is supported on Solr only. You provide the field (from one of the `SearchIndex` classes) you would like stats on.

In the search results you get back, stats will be populated in the `SearchResult` object. You can access them via the `stats_results` method.

Example:

```
# Get stats on the author field.
SearchQuerySet().filter(content='foo').stats('author')
```

stats_facet

SearchQuerySet.stats_facet(self, field,

Adds stats facet for the given field and facet_fields represents the faceted fields. This is supported on Solr only.

Example:

```
# Get stats on the author field, and stats on the author field
# faceted by bookstore.
SearchQuerySet().filter(content='foo').stats_facet('author', 'bookstore')
```

distance

SearchQuerySet.distance(self, field, point):

Spatial: Denotes results must have distance measurements from the provided point.

See the *Spatial Search* docs for more information.

narrow

SearchQuerySet.narrow(self, query)

Pulls a subset of documents from the search engine to search within. This is for advanced usage, especially useful when faceting.

Example:

```
# Search, from recipes containing 'blend', for recipes containing 'banana'.
SearchQuerySet().narrow('blend').filter(content='banana')

# Using a fielded search where the recipe's title contains 'smoothie', find all recipes published before 2009.
SearchQuerySet().narrow('title:smoothie').filter(pub_date__lte=datetime.datetime(2009, 1, 1))
```

By using narrow, you can create drill-down interfaces for faceting by applying narrow calls for each facet that gets selected.

This method is different from SearchQuerySet.filter() in that it does not affect the query sent to the engine. It pre-limits the document set being searched. Generally speaking, if you're in doubt of whether to use filter or narrow, use filter.

Note: This method is, generally speaking, not necessarily portable between backends. The syntax is entirely dependent on the backend, though most backends have a similar syntax for basic fielded queries. No validation/cleansing is performed and it is up to the developer to ensure the query's syntax is correct.

raw_search

SearchQuerySet.raw_search(self, query_string, **kwargs)

Passes a raw query directly to the backend. This is for advanced usage, where the desired query can not be expressed via `SearchQuerySet`.

This method is still supported, however it now uses the much more flexible `Raw` input type (*Input Types*).

Warning: Different from Haystack 1.X, this method no longer causes immediate evaluation & now chains appropriately.

Example:

```
# In the case of Solr... (this example could be expressed with SearchQuerySet)
SearchQuerySet().raw_search('django_ct:blog.blogentry "However, it is"')

# Equivalent.
from haystack.inputs import Raw
sqs = SearchQuerySet().filter(content=Raw('django_ct:blog.blogentry "However, it is"'))
```

Please note that this is **NOT** portable between backends. The syntax is entirely dependent on the backend. No validation/cleansing is performed and it is up to the developer to ensure the query's syntax is correct.

Further, the use of `**kwargs` are completely undocumented intentionally. If a third-party backend can implement special features beyond what's present, it should use those `**kwargs` for passing that information. Developers should be careful to make sure there are no conflicts with the backend's `search` method, as that is called directly.

`load_all`

`SearchQuerySet.load_all(self)`

Efficiently populates the objects in the search results. Without using this method, DB lookups are done on a per-object basis, resulting in many individual trips to the database. If `load_all` is used, the `SearchQuerySet` will group similar objects into a single query, resulting in only as many queries as there are different object types returned.

Example:

```
SearchQuerySet().filter(content='foo').load_all()
```

`load_all_queryset`

`SearchQuerySet.load_all_queryset(self, model_class, queryset)`

Deprecated for removal before Haystack 1.0-final.

Please see the docs on `RelatedSearchQuerySet`.

`auto_query`

`SearchQuerySet.auto_query(self, query_string, fieldname=None)`

Performs a best guess constructing the search query.

This method is intended for common use directly with a user's query. This method is still supported, however it now uses the much more flexible `AutoQuery` input type (*Input Types*).

It handles exact matches (specified with single or double quotes), negation (using a `-` immediately before the term) and joining remaining terms with the operator specified in `HAYSTACK_DEFAULT_OPERATOR`.

Example:

```
sqs = SearchQuerySet().auto_query('goldfish "old one eye" -tank')

# Equivalent.
from haystack.inputs import AutoQuery
sqs = SearchQuerySet().filter(content=AutoQuery('goldfish "old one eye" -tank'))

# Against a different field.
sqs = SearchQuerySet().filter(title=AutoQuery('goldfish "old one eye" -tank'))
```

autocomplete

A shortcut method to perform an autocomplete search.

Must be run against fields that are either `NgramField` or `EdgeNgramField`.

Example:

```
SearchQuerySet().autocomplete(title_autocomplete='gol')
```

more_like_this

`SearchQuerySet.more_like_this` (*self*, *model_instance*)

Finds similar results to the object passed in.

You should pass in an instance of a model (for example, one fetched via a `get` in Django's ORM). This will execute a query on the backend that searches for similar results. The instance you pass in should be an indexed object. Previously called methods will have an effect on the provided results.

It will evaluate its own backend-specific query and populate the `SearchQuerySet` in the same manner as other methods.

Example:

```
entry = Entry.objects.get(slug='haystack-one-oh-released')
mlt = SearchQuerySet().more_like_this(entry)
mlt.count() # 5
mlt[0].object.title # "Haystack Beta 1 Released"

# ...or...
mlt = SearchQuerySet().filter(public=True).exclude(pub_date__lte=datetime.date(2009, 7, 21)).more_like_this(entry)
mlt.count() # 2
mlt[0].object.title # "Haystack Beta 1 Released"
```

using

`SearchQuerySet.using` (*self*, *connection_name*)

Allows switching which connection the `SearchQuerySet` uses to search in.

Example:

```
# Let the routers decide which connection to use.
sqs = SearchQuerySet().all()

# Specify the 'default'.
sqs = SearchQuerySet().all().using('default')
```

Methods That Do Not Return A SearchQuerySet

count

SearchQuerySet.**count** (*self*)

Returns the total number of matching results.

This returns an integer count of the total number of results the search backend found that matched. This method causes the query to evaluate and run the search.

Example:

```
SearchQuerySet().filter(content='foo').count()
```

best_match

SearchQuerySet.**best_match** (*self*)

Returns the best/top search result that matches the query.

This method causes the query to evaluate and run the search. This method returns a SearchResult object that is the best match the search backend found:

```
foo = SearchQuerySet().filter(content='foo').best_match()
foo.id # Something like 5.

# Identical to:
foo = SearchQuerySet().filter(content='foo')[0]
```

latest

SearchQuerySet.**latest** (*self*, *date_field*)

Returns the most recent search result that matches the query.

This method causes the query to evaluate and run the search. This method returns a SearchResult object that is the most recent match the search backend found:

```
foo = SearchQuerySet().filter(content='foo').latest('pub_date')
foo.id # Something like 3.

# Identical to:
foo = SearchQuerySet().filter(content='foo').order_by('-pub_date')[0]
```

facet_counts

SearchQuerySet.**facet_counts** (*self*)

Returns the facet counts found by the query. This will cause the query to execute and should generally be used when presenting the data (template-level).

You receive back a dictionary with three keys: `fields`, `dates` and `queries`. Each contains the facet counts for whatever facets you specified within your SearchQuerySet.

Note: The resulting dictionary may change before 1.0 release. It's fairly backend-specific at the time of writing. Standardizing is waiting on implementing other backends that support faceting and ensuring that the results presented will meet their needs as well.

Example:

```
# Count document hits for each author.
sqs = SearchQuerySet().filter(content='foo').facet('author')

sqs.facet_counts()
# Gives the following response:
# {
#   'dates': {},
#   'fields': {
#     'author': [
#       ('john', 4),
#       ('daniel', 2),
#       ('sally', 1),
#       ('terry', 1),
#     ],
#   },
#   'queries': {}
# }
```

stats_results

SearchQuerySet.stats_results(self):

Returns the stats results found by the query.

This will cause the query to execute and should generally be used when presenting the data (template-level).

You receive back a dictionary with three keys: `fields`, `dates` and `queries`. Each contains the facet counts for whatever facets you specified within your `SearchQuerySet`.

Note: The resulting dictionary may change before 1.0 release. It's fairly backend-specific at the time of writing. Standardizing is waiting on implementing other backends that support faceting and ensuring that the results presented will meet their needs as well.

Example:

```
# Count document hits for each author.
sqs = SearchQuerySet().filter(content='foo').stats('price')

sqs.stats_results()

# Gives the following response
# {
#   'stats_fields':{
#     'author':{
#       'min': 0.0,
#       'max': 2199.0,
#       'sum': 5251.2699999999995,
#     }
#   }
# }
```

```
#         'count': 15,  
#         'missing': 11,  
#         'sumOfSquares': 6038619.160300001,  
#         'mean': 350.08466666666664,  
#         'stddev': 547.737557906113  
#     }  
# }  
# }  
# }
```

spelling_suggestion

`SearchQuerySet.spelling_suggestion` (*self*, *preferred_query=None*)

Returns the spelling suggestion found by the query.

To work, you must set `INCLUDE_SPELLING` within your connection's settings dictionary to `True`. Otherwise, `None` will be returned.

This method causes the query to evaluate and run the search if it hasn't already run. Search results will be populated as normal but with an additional spelling suggestion. Note that this does *NOT* run the revised query, only suggests improvements.

If provided, the optional argument to this method lets you specify an alternate query for the spelling suggestion to be run on. This is useful for passing along a raw user-provided query, especially when there are many methods chained on the `SearchQuerySet`.

Example:

```
sqs = SearchQuerySet().auto_query('mor exmples')  
sqs.spelling_suggestion() # u'more examples'  
  
# ...or...  
suggestion = SearchQuerySet().spelling_suggestion('moar exmples')  
suggestion # u'more examples'
```

values

`SearchQuerySet.values` (*self*, **fields*)

Returns a list of dictionaries, each containing the key/value pairs for the result, exactly like Django's `ValuesQuerySet`.

This method causes the query to evaluate and run the search if it hasn't already run.

You must provide a list of one or more fields as arguments. These fields will be the ones included in the individual results.

Example:

```
sqs = SearchQuerySet().auto_query('banana').values('title', 'description')
```

values_list

`SearchQuerySet.values_list` (*self*, **fields*, ***kwargs*)

Returns a list of field values as tuples, exactly like Django's `ValuesListQuerySet`.

This method causes the query to evaluate and run the search if it hasn't already run.

You must provide a list of one or more fields as arguments. These fields will be the ones included in the individual results.

You may optionally also provide a `flat=True` kwarg, which in the case of a single field being provided, will return a flat list of that field rather than a list of tuples.

Example:

```
sqs = SearchQuerySet().auto_query('banana').values_list('title', 'description')

# ...or just the titles as a flat list...
sqs = SearchQuerySet().auto_query('banana').values_list('title', flat=True)
```

Field Lookups

The following lookup types are supported:

- contains
- exact
- gt
- gte
- lt
- lte
- in
- startswith
- range

These options are similar in function to the way Django's lookup types work. The actual behavior of these lookups is backend-specific.

Warning: The `startswith` filter is strongly affected by the other ways the engine parses data, especially in regards to stemming (see [Glossary](#)). This can mean that if the query ends in a vowel or a plural form, it may get stemmed before being evaluated.

This is both backend-specific and yet fairly consistent between engines, and may be the cause of sometimes unexpected results.

Warning: The `contains` filter became the new default filter as of Haystack v2.X (the default in Haystack v1.X was `exact`). This changed because `exact` caused problems and was unintuitive for new people trying to use Haystack. `contains` is a much more natural usage.

If you had an app built on Haystack v1.X & are upgrading, you'll need to sanity-check & possibly change any code that was relying on the default. The solution is just to add `__exact` to any "bare" field in a `.filter(...)` clause.

Example:

```
SearchQuerySet().filter(content='foo')

# Identical to:
```

```

SearchQuerySet().filter(content__contains='foo')

# Phrase matching.
SearchQuerySet().filter(content__exact='hello world')

# Other usages look like:
SearchQuerySet().filter(pub_date__gte=datetime.date(2008, 1, 1), pub_date__lt=datetime.date(2009, 1, 1))
SearchQuerySet().filter(author__in=['daniel', 'john', 'jane'])
SearchQuerySet().filter(view_count__range=[3, 5])

```

3.1.6 EmptySearchQuerySet

Also included in Haystack is an `EmptySearchQuerySet` class. It behaves just like `SearchQuerySet` but will always return zero results. This is useful for places where you want no query to occur or results to be returned.

3.1.7 RelatedSearchQuerySet

Sometimes you need to filter results based on relations in the database that are not present in the search index or are difficult to express that way. To this end, `RelatedSearchQuerySet` allows you to post-process the search results by calling `load_all_queryset`.

Warning: `RelatedSearchQuerySet` can have negative performance implications. Because results are excluded based on the database after the search query has been run, you can't guarantee offsets within the cache. Therefore, the entire cache that appears before the offset you request must be filled in order to produce consistent results. On large result sets and at higher slices, this can take time. This is the old behavior of `SearchQuerySet`, so performance is no worse than the early days of Haystack.

It supports all other methods that the standard `SearchQuerySet` does, with the addition of the `load_all_queryset` method and paying attention to the `load_all_queryset` method of `SearchIndex` objects when populating the cache.

`load_all_queryset`

```
RelatedSearchQuerySet.load_all_queryset(self, model_class, queryset)
```

Allows for specifying a custom `QuerySet` that changes how `load_all` will fetch records for the provided model. This is useful for post-processing the results from the query, enabling things like adding `select_related` or filtering certain data.

Example:

```

sqs = RelatedSearchQuerySet().filter(content='foo').load_all()
# For the Entry model, we want to include related models directly associated
# with the Entry to save on DB queries.
sqs = sqs.load_all_queryset(Entry, Entry.objects.all().select_related(depth=1))

```

This method chains indefinitely, so you can specify `QuerySets` for as many models as you wish, one per model. The `SearchQuerySet` appends on a call to `in_bulk`, so be sure that the `QuerySet` you provide can accommodate this and that the ids passed to `in_bulk` will map to the model in question.

If you need to do this frequently and have one `QuerySet` you'd like to apply everywhere, you can specify this at the `SearchIndex` level using the `load_all_queryset` method. See [SearchIndex API](#) for usage.

3.2 SearchIndex API

class SearchIndex

The `SearchIndex` class allows the application developer a way to provide data to the backend in a structured format. Developers familiar with Django's `Form` or `Model` classes should find the syntax for indexes familiar.

This class is arguably the most important part of integrating Haystack into your application, as it has a large impact on the quality of the search results and how easy it is for users to find what they're looking for. Care and effort should be put into making your indexes the best they can be.

3.2.1 Quick Start

For the impatient:

```
import datetime
from haystack import indexes
from myapp.models import Note

class NoteIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    author = indexes.CharField(model_attr='user')
    pub_date = indexes.DateTimeField(model_attr='pub_date')

    def get_model(self):
        return Note

    def index_queryset(self, using=None):
        """Used when the entire index for model is updated."""
        return self.get_model().objects.filter(pub_date__lte=datetime.datetime.now())
```

3.2.2 Background

Unlike relational databases, most search engines supported by Haystack are primarily document-based. They focus on a single text blob which they tokenize, analyze and index. When searching, this field is usually the primary one that is searched.

Further, the schema used by most engines is the same for all types of data added, unlike a relational database that has a table schema for each chunk of data.

It may be helpful to think of your search index as something closer to a key-value store instead of imagining it in terms of a RDBMS.

Why Create Fields?

Despite being primarily document-driven, most search engines also support the ability to associate other relevant data with the indexed document. These attributes can be mapped through the use of fields within Haystack.

Common uses include storing pertinent data information, categorizations of the document, author information and related data. By adding fields for these pieces of data, you provide a means to further narrow/filter search terms. This can be useful from either a UI perspective (a better advanced search form) or from a developer standpoint (section-dependent search, off-loading certain tasks to search, et cetera).

Warning: Haystack reserves the following field names for internal use: `id`, `django_ct`, `django_id` & `content`. The name & type names used to be reserved but no longer are. You can override these field names using the `HAYSTACK_ID_FIELD`, `HAYSTACK_DJANGO_CT_FIELD` & `HAYSTACK_DJANGO_ID_FIELD` if needed.

Significance Of `document=True`

Most search engines that were candidates for inclusion in Haystack all had a central concept of a document that they indexed. These documents form a corpus within which to primarily search. Because this ideal is so central and most of Haystack is designed to have pluggable backends, it is important to ensure that all engines have at least a bare minimum of the data they need to function.

As a result, when creating a `SearchIndex`, at least one field must be marked with `document=True`. This signifies to Haystack that whatever is placed in this field while indexing is to be the primary text the search engine indexes. The name of this field can be almost anything, but `text` is one of the more common names used.

Stored/Indexed Fields

One shortcoming of the use of search is that you rarely have all or the most up-to-date information about an object in the index. As a result, when retrieving search results, you will likely have to access the object in the database to provide better information.

However, this can also hit the database quite heavily (think `.get(pk=result.id)` per object). If your search is popular, this can lead to a big performance hit. There are two ways to prevent this. The first way is `SearchQuerySet.load_all`, which tries to group all similar objects and pull them through one query instead of many. This still hits the DB and incurs a performance penalty.

The other option is to leverage stored fields. By default, all fields in Haystack are both indexed (searchable by the engine) and stored (retained by the engine and presented in the results). By using a stored field, you can store commonly used data in such a way that you don't need to hit the database when processing the search result to get more information.

For example, one great way to leverage this is to pre-rendering an object's search result template DURING indexing. You define an additional field, render a template with it and it follows the main indexed record into the index. Then, when that record is pulled when it matches a query, you can simply display the contents of that field, which avoids the database hit:

Within `myapp/search_indexes.py`:

```
class NoteIndex(SearchIndex, indexes.Indexable):
    text = CharField(document=True, use_template=True)
    author = CharField(model_attr='user')
    pub_date = DateTimeField(model_attr='pub_date')
    # Define the additional field.
    rendered = CharField(use_template=True, indexed=False)
```

Then, inside a template named `search/indexes/myapp/note_rendered.txt`:

```
<h2>{{ object.title }}</h2>
<p>{{ object.content }}</p>
```

And finally, in `search/search.html`:

```

...
{% for result in page.object_list %}
  <div class="search_result">
    {{ result.rendered|safe }}
  </div>
{% endfor %}

```

3.2.3 Keeping The Index Fresh

There are several approaches to keeping the search index in sync with your database. None are more correct than the others and depending the traffic you see, the churn rate of your data and what concerns are important to you (CPU load, how recent, et cetera).

The conventional method is to use `SearchIndex` in combination with cron jobs. Running a `./manage.py update_index` every couple hours will keep your data in sync within that timeframe and will handle the updates in a very efficient batch. Additionally, Whoosh (and to a lesser extent Xapian) behave better when using this approach.

Another option is to use `RealtimeSignalProcessor`, which uses Django's signals to immediately update the index any time a model is saved/deleted. This yields a much more current search index at the expense of being fairly inefficient. Solr & Elasticsearch are the only backends that handles this well under load, and even then, you should make sure you have the server capacity to spare.

A third option is to develop a custom `QueuedSignalProcessor` that, much like `RealtimeSignalProcessor`, uses Django's signals to enqueue messages for updates/deletes. Then writing a management command to consume these messages in batches, yielding a nice compromise between the previous two options.

Note: Haystack doesn't ship with a `QueuedSignalProcessor` largely because there is such a diversity of lightweight queuing options and that they tend to polarize developers. Queuing is outside of Haystack's goals (provide good, powerful search) and, as such, is left to the developer.

Additionally, the implementation is relatively trivial & there are already good third-party add-ons for Haystack to enable this.

3.2.4 Advanced Data Preparation

In most cases, using the `model_attr` parameter on your fields allows you to easily get data from a Django model to the document in your index, as it handles both direct attribute access as well as callable functions within your model.

Note: The `model_attr` keyword argument also can look through relations in models. So you can do something like `model_attr='author__first_name'` to pull just the first name of the author, similar to some lookups used by Django's ORM.

However, sometimes, even more control over what gets placed in your index is needed. To facilitate this, `SearchIndex` objects have a 'preparation' stage that populates data just before it is indexed. You can hook into this phase in several ways.

This should be very familiar to developers who have used Django's `forms` before as it loosely follows similar concepts, though the emphasis here is less on cleansing data from user input and more on making the data friendly to the search backend.

1. prepare_FOO(self, object)

The most common way to affect a single field's data is to create a `prepare_FOO` method (where FOO is the name of the field). As a parameter to this method, you will receive the instance that is attempting to be indexed.

Note: This method is analogous to Django's `Form.clean_FOO` methods.

To keep with our existing example, one use case might be altering the name inside the `author` field to be "firstname lastname <email>". In this case, you might write the following code:

```
class NoteIndex(SearchIndex, indexes.Indexable):
    text = CharField(document=True, use_template=True)
    author = CharField(model_attr='user')
    pub_date = DateTimeField(model_attr='pub_date')

    def get_model(self):
        return Note

    def prepare_author(self, obj):
        return "%s <%s>" % (obj.user.get_full_name(), obj.user.email)
```

This method should return a single value (or list/tuple/dict) to populate that field's data upon indexing. Note that this method takes priority over whatever data may come from the field itself.

Just like `Form.clean_FOO`, the field's `prepare` runs before the `prepare_FOO`, allowing you to access `self.prepared_data`. For example:

```
class NoteIndex(SearchIndex, indexes.Indexable):
    text = CharField(document=True, use_template=True)
    author = CharField(model_attr='user')
    pub_date = DateTimeField(model_attr='pub_date')

    def get_model(self):
        return Note

    def prepare_author(self, obj):
        # Say we want last name first, the hard way.
        author = u''

        if 'author' in self.prepared_data:
            name_bits = self.prepared_data['author'].split()
            author = "%s, %s" % (name_bits[-1], ' '.join(name_bits[:-1]))

        return author
```

This method is fully function with `model_attr`, so if there's no convenient way to access the data you want, this is an excellent way to prepare it:

```
class NoteIndex(SearchIndex, indexes.Indexable):
    text = CharField(document=True, use_template=True)
    author = CharField(model_attr='user')
    categories = MultiValueField()
    pub_date = DateTimeField(model_attr='pub_date')

    def get_model(self):
        return Note

    def prepare_categories(self, obj):
```

```
# Since we're using a M2M relationship with a complex lookup,
# we can prepare the list here.
return [category.id for category in obj.category_set.active().order_by('-created')]
```

2. prepare(self, object)

Each `SearchIndex` gets a `prepare` method, which handles collecting all the data. This method should return a dictionary that will be the final data used by the search backend.

Overriding this method is useful if you need to collect more than one piece of data or need to incorporate additional data that is not well represented by a single `SearchField`. An example might look like:

```
class NoteIndex(SearchIndex, indexes.Indexable):
    text = CharField(document=True, use_template=True)
    author = CharField(model_attr='user')
    pub_date = DateTimeField(model_attr='pub_date')

    def get_model(self):
        return Note

    def prepare(self, object):
        self.prepared_data = super(NoteIndex, self).prepare(object)

        # Add in tags (assuming there's a M2M relationship to Tag on the model).
        # Note that this would NOT get picked up by the automatic
        # schema tools provided by Haystack.
        self.prepared_data['tags'] = [tag.name for tag in object.tags.all()]

    return self.prepared_data
```

If you choose to use this method, you should make a point to be careful to call the `super()` method before altering the data. Without doing so, you may have an incomplete set of data populating your indexes.

This method has the final say in all data, overriding both what the fields provide as well as any `prepare_FOO` methods on the class.

Note: This method is roughly analogous to Django's `Form.full_clean` and `Form.clean` methods. However, unlike these methods, it is not fired as the result of trying to access `self.prepared_data`. It requires an explicit call.

3. Overriding prepare(self, object) On Individual SearchField Objects

The final way to manipulate your data is to implement a custom `SearchField` object and write its `prepare` method to populate/alter the data any way you choose. For instance, a (naive) user-created `GeoPointField` might look something like:

```
from haystack import indexes

class GeoPointField(indexes.CharField):
    def __init__(self, **kwargs):
        kwargs['default'] = '0.00-0.00'
        super(GeoPointField, self).__init__(**kwargs)
```

```
def prepare(self, obj):  
    return unicode("%s-%s" % (obj.latitude, obj.longitude))
```

The `prepare` method simply returns the value to be used for that field. It's entirely possible to include data that's not directly referenced to the object here, depending on your needs.

Note that this is NOT a recommended approach to storing geographic data in a search engine (there is no formal suggestion on this as support is usually non-existent), merely an example of how to extend existing fields.

Note: This method is analagous to Django's `Field.clean` methods.

3.2.5 Adding New Fields

If you have an existing `SearchIndex` and you add a new field to it, Haystack will add this new data on any updates it sees after that point. However, this will not populate the existing data you already have.

In order for the data to be picked up, you will need to run `./manage.py rebuild_index`. This will cause all backends to rebuild the existing data already present in the quickest and most efficient way.

Note: With the Solr backend, you'll also have to add to the appropriate `schema.xml` for your configuration before running the `rebuild_index`.

3.2.6 Search Index

`get_model`

`SearchIndex.get_model(self)`

Should return the `Model` class (not an instance) that the rest of the `SearchIndex` should use.

This method is required & you must override it to return the correct class.

`index_queryset`

`SearchIndex.index_queryset(self, using=None)`

Get the default `QuerySet` to index when doing a full update.

Subclasses can override this method to avoid indexing certain objects.

`read_queryset`

`SearchIndex.read_queryset(self, using=None)`

Get the default `QuerySet` for read actions.

Subclasses can override this method to work with other managers. Useful when working with default managers that filter some objects.

build_queryset

`SearchIndex.build_queryset` (*self*, *start_date=None*, *end_date=None*)

Get the default QuerySet to index when doing an index update.

Subclasses can override this method to take into account related model modification times.

The default is to use `SearchIndex.index_queryset` and filter based on `SearchIndex.get_updated_field`

prepare

`SearchIndex.prepare` (*self*, *obj*)

Fetches and adds/alters data before indexing.

get_content_field

`SearchIndex.get_content_field` (*self*)

Returns the field that supplies the primary document to be indexed.

update

`SearchIndex.update` (*self*, *using=None*)

Updates the entire index.

If `using` is provided, it specifies which connection should be used. Default relies on the routers to decide which backend should be used.

update_object

`SearchIndex.update_object` (*self*, *instance*, *using=None*, ***kwargs*)

Update the index for a single object. Attached to the class's post-save hook.

If `using` is provided, it specifies which connection should be used. Default relies on the routers to decide which backend should be used.

remove_object

`SearchIndex.remove_object` (*self*, *instance*, *using=None*, ***kwargs*)

Remove an object from the index. Attached to the class's post-delete hook.

If `using` is provided, it specifies which connection should be used. Default relies on the routers to decide which backend should be used.

`clear`

`SearchIndex.clear` (*self*, *using=None*)

Clears the entire index.

If *using* is provided, it specifies which connection should be used. Default relies on the routers to decide which backend should be used.

`reindex`

`SearchIndex.reindex` (*self*, *using=None*)

Completely clears the index for this model and rebuilds it.

If *using* is provided, it specifies which connection should be used. Default relies on the routers to decide which backend should be used.

`get_updated_field`

`SearchIndex.get_updated_field` (*self*)

Get the field name that represents the updated date for the model.

If specified, this is used by the `reindex` command to filter out results from the `QuerySet`, enabling you to reindex only recent records. This method should either return `None` (reindex everything always) or a string of the Model's `DateField/DateTimeField` name.

`should_update`

`SearchIndex.should_update` (*self*, *instance*, ***kwargs*)

Determine if an object should be updated in the index.

It's useful to override this when an object may save frequently and cause excessive reindexing. You should check conditions on the instance and return `False` if it is not to be indexed.

The *kwargs* passed along to this method can be the same as the ones passed by Django when a Model is saved/delete, so it's possible to check if the object has been created or not. See `django.db.models.signals.post_save` for details on what is passed.

By default, returns `True` (always reindex).

`load_all_queryset`

`SearchIndex.load_all_queryset` (*self*)

Provides the ability to override how objects get loaded in conjunction with `RelatedSearchQuerySet.load_all`. This is useful for post-processing the results from the query, enabling things like adding `select_related` or filtering certain data.

Warning: Utilizing this functionality can have negative performance implications. Please see the section on `RelatedSearchQuerySet` within [SearchQuerySet API](#) for further information.

By default, returns `all()` on the model's default manager.

Example:

```

class NoteIndex(SearchIndex, indexes.Indexable):
    text = CharField(document=True, use_template=True)
    author = CharField(model_attr='user')
    pub_date = DateTimeField(model_attr='pub_date')

    def get_model(self):
        return Note

    def load_all_queryset(self):
        # Pull all objects related to the Note in search results.
        return Note.objects.all().select_related()

```

When searching, the `RelatedSearchQuerySet` appends on a call to `in_bulk`, so be sure that the `QuerySet` you provide can accommodate this and that the `ids` passed to `in_bulk` will map to the model in question.

If you need a specific `QuerySet` in one place, you can specify this at the `RelatedSearchQuerySet` level using the `load_all_queryset` method. See [SearchQuerySet API](#) for usage.

3.2.7 ModelSearchIndex

The `ModelSearchIndex` class allows for automatic generation of a `SearchIndex` based on the fields of the model assigned to it.

With the exception of the automated introspection, it is a `SearchIndex` class, so all notes above pertaining to `SearchIndexes` apply. As with the `ModelForm` class in Django, it employs an inner class called `Meta`, which should either contain a `pass` to include all fields, a `fields` list to specify a whitelisted set of fields or `excludes` to prevent certain fields from appearing in the class. Unlike `ModelForm`, you should **NOT** specify a `model` attribute, as that is already handled when registering the class.

In addition, it adds a `text` field that is the `document=True` field and has `use_template=True` option set, just like the `BasicSearchIndex`.

Warning: Usage of this class might result in inferior `SearchIndex` objects, which can directly affect your search results. Use this to establish basic functionality and move to custom `SearchIndex` objects for better control.

At this time, it does not handle related fields.

Quick Start

For the impatient:

```

import datetime
from haystack import indexes
from myapp.models import Note

# All Fields
class AllNoteIndex(indexes.ModelSearchIndex, indexes.Indexable):
    class Meta:
        model = Note

# Blacklisted Fields
class LimitedNoteIndex(indexes.ModelSearchIndex, indexes.Indexable):
    class Meta:
        model = Note
        excludes = ['user']

```

```
# Whitelisted Fields
class NoteIndex(indexes.ModelSearchIndex, indexes.Indexable):
    class Meta:
        model = Note
        fields = ['user', 'pub_date']

    # Note that regular ``SearchIndex`` methods apply.
    def index_queryset(self, using=None):
        "Used when the entire index for model is updated."
        return Note.objects.filter(pub_date__lte=datetime.datetime.now())
```

3.3 Input Types

Input types allow you to specify more advanced query behavior. They serve as a way to alter the query, often in backend-specific ways, without altering your Python code; as well as enabling use of more advanced features.

Input types currently are only useful with the `filter/exclude` methods on `SearchQuerySet`. Expanding this support to other methods is on the roadmap.

3.3.1 Available Input Types

Included with Haystack are the following input types:

Raw

class `haystack.inputs.Raw`

Raw allows you to specify backend-specific query syntax. If Haystack doesn't provide a way to access special query functionality, you can make use of this input type to pass it along.

Example:

```
# Fielded.
sqs = SearchQuerySet().filter(author=Raw('daniel OR jones'))

# Non-fielded.
# See ``AltParser`` for a better way to construct this.
sqs = SearchQuerySet().filter(content=Raw('{!dismax qf=author mm=1}haystack'))
```

Clean

class `haystack.inputs.Clean`

Clean takes standard user (untrusted) input and sanitizes it. It ensures that no unintended operators or special characters make it into the query.

This is roughly analogous to Django's autoescape support.

Note: By default, if you hand a `SearchQuerySet` a bare string, it will get wrapped in this class.

Example:

```
# This becomes "daniel or jones".
sqs = SearchQuerySet().filter(content=Clean('daniel OR jones'))

# Things like ``:`` & ``/`` get escaped.
sqs = SearchQuerySet().filter(url=Clean('http://www.example.com'))

# Equivalent (automatically wrapped in ``Clean``).
sqs = SearchQuerySet().filter(url='http://www.example.com')
```

Exact

class haystack.inputs.**Exact**

Exact allows for making sure a phrase is exactly matched, unlike the usual AND lookups, where words may be far apart.

Example:

```
sqs = SearchQuerySet().filter(author=Exact('n-gram support'))

# Equivalent.
sqs = SearchQuerySet().filter(author__exact='n-gram support')
```

Not

class haystack.inputs.**Not**

Not allows negation of the query fragment it wraps. As **Not** is a subclass of **Clean**, it will also sanitize the query.

This is generally only used internally. Most people prefer to use the `SearchQuerySet.exclude` method.

Example:

```
sqs = SearchQuerySet().filter(author=Not('daniel'))
```

AutoQuery

class haystack.inputs.**AutoQuery**

AutoQuery takes a more complex user query (that includes simple, standard query syntax bits) & forms a proper query out of them. It also handles sanitizing that query using **Clean** to ensure the query doesn't break.

AutoQuery accommodates for handling regular words, NOT-ing words & extracting exact phrases.

Example:

```
# Against the main text field with an accidental ":" before "search".
# Generates a query like ``haystack (NOT whoosh) "fast search"``
sqs = SearchQuerySet().filter(content=AutoQuery('haystack -whoosh "fast :search"'))

# Equivalent.
sqs = SearchQuerySet().auto_query('haystack -whoosh "fast :search"')

# Fielded.
sqs = SearchQuerySet().filter(author=AutoQuery('daniel -day -lewis'))
```

AltParser

class haystack.inputs.AltParser

AltParser lets you specify that a portion of the query should use a separate parser in the search engine. This is search-engine-specific, so it may decrease the portability of your app.

Currently only supported under Solr.

Example:

```
# DisMax.
sqs = SearchQuerySet().filter(content=AltParser('dismax', 'haystack', qf='text', mm=1))

# Prior to the spatial support, you could do...
sqs = SearchQuerySet().filter(content=AltParser('dismax', 'haystack', qf='author', mm=1))
```

3.3.2 Creating Your Own Input Types

Building your own input type is relatively simple. All input types are simple classes that provide an `__init__` & a `prepare` method.

The `__init__` may accept any `args`/`kwargs`, though the typical use usually just involves a query string.

The `prepare` method lets you alter the query the user provided before it becomes of the main query. It is lazy, called as late as possible, right before the final query is built & shipped to the engine.

A full, if somewhat silly, example look like:

```
from haystack.inputs import Clean

class NoShoutCaps(Clean):
    input_type_name = 'no_shout_caps'
    # This is the default & doesn't need to be specified.
    post_process = True

    def __init__(self, query_string, **kwargs):
        # Stash the original, if you need it.
        self.original = query_string
        super(NoShoutCaps, self).__init__(query_string, **kwargs)

    def prepare(self, query_obj):
        # We need a reference to the current ``SearchQuery`` object this
        # will run against, in case we need backend-specific code.
        query_string = super(NoShoutCaps, self).prepare(query_obj)

        # Take that, capital letters!
        return query_string.lower()
```

3.4 SearchField API

class SearchField

The `SearchField` and its subclasses provides a way to declare what data you're interested in indexing. They are used with `SearchIndexes`, much like `forms.*Field` are used within forms or `models.*Field` within models.

They provide both the means for storing data in the index, as well as preparing the data before it's placed in the index. Haystack uses all fields from all `SearchIndex` classes to determine what the engine's index schema ought to look like.

In practice, you'll likely never actually use the base `SearchField`, as the subclasses are much better at handling real data.

3.4.1 Subclasses

Included with Haystack are the following field types:

- `BooleanField`
- `CharField`
- `DateField`
- `DateTimeField`
- `DecimalField`
- `EdgeNgramField`
- `FloatField`
- `IntegerField`
- `LocationField`
- `MultiValueField`
- `NgramField`

And equivalent faceted versions:

- `FacetBooleanField`
- `FacetCharField`
- `FacetDateField`
- `FacetDateTimeField`
- `FacetDecimalField`
- `FacetFloatField`
- `FacetIntegerField`
- `FacetMultiValueField`

Note: There is no faceted variant of the n-gram fields. Because of how the engine generates n-grams, faceting on these field types (`NgramField` & `EdgeNgram`) would make very little sense.

3.4.2 Usage

While `SearchField` objects can be used on their own, they're generally used within a `SearchIndex`. You use them in a declarative manner, just like fields in `django.forms.Form` or `django.db.models.Model` objects. For example:

```
from haystack import indexes
from myapp.models import Note

class NoteIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    author = indexes.CharField(model_attr='user')
    pub_date = indexes.DateTimeField(model_attr='pub_date')

    def get_model(self):
        return Note
```

This will hook up those fields with the index and, when updating a `Model` object, pull the relevant data out and prepare it for storage in the index.

3.4.3 Field Options

default

`SearchField.default`

Provides a means for specifying a fallback value in the event that no data is found for the field. Can be either a value or a callable.

document

`SearchField.document`

A boolean flag that indicates which of the fields in the `SearchIndex` ought to be the primary field for searching within. Default is `False`.

Note: Only one field can be marked as the `document=True` field, so you should standardize this name and the format of the field between all of your `SearchIndex` classes.

indexed

`SearchField.indexed`

A boolean flag for indicating whether or not the the data from this field will be searchable within the index. Default is `True`.

The companion of this option is `stored`.

index_fieldname

`SearchField.index_fieldname`

The `index_fieldname` option allows you to force the name of the field in the index. This does not change how Haystack refers to the field. This is useful when using Solr's dynamic attributes or when integrating with other external software.

Default is variable name of the field within the `SearchIndex`.

model_attr**SearchField.model_attr**

The `model_attr` option is a shortcut for preparing data. Rather than having to manually fetch data out of a `Model`, `model_attr` allows you to specify a string that will automatically pull data out for you. For example:

```
# Automatically looks within the model and populates the field with
# the ``last_name`` attribute.
author = CharField(model_attr='last_name')
```

It also handles callables:

```
# On a ``User`` object, pulls the full name as pieced together by the
# ``get_full_name`` method.
author = CharField(model_attr='get_full_name')
```

And can look through relations:

```
# Pulls the ``bio`` field from a ``UserProfile`` object that has a
# ``OneToOneField`` relationship to a ``User`` object.
biography = CharField(model_attr='user__profile__bio')
```

null**SearchField.null**

A boolean flag for indicating whether or not it's permissible for the field not to contain any data. Default is `False`.

Note: Unlike Django's database layer, which injects a `NULL` into the database when a field is marked nullable, `null=True` will actually exclude that field from being included with the document. This more efficient for the search engine to deal with.

stored**SearchField.stored**

A boolean flag for indicating whether or not the data from this field will be stored within the index. Default is `True`.

This is useful for pulling data out of the index along with the search result in order to save on hits to the database.

The companion of this option is `indexed`.

template_name**SearchField.template_name**

Allows you to override the name of the template to use when preparing data. By default, the data templates for fields are located within your `TEMPLATE_DIRS` under a path like `search/indexes/{app_label}/{model_name}_{field_name}.txt`. This option lets you override that path (though still within `TEMPLATE_DIRS`).

Example:

```
bio = CharField(use_template=True, template_name='myapp/data/bio.txt')
```

You can also provide a list of templates, as `loader.select_template` is used under the hood.

Example:

```
bio = CharField(use_template=True, template_name=['myapp/data/bio.txt', 'myapp/bio.txt', 'bio.txt'])
```

`use_template`

`SearchField.use_template`

A boolean flag for indicating whether or not a field should prepare its data via a data template or not. Default is `False`.

Data templates are extremely useful, as they let you easily tie together different parts of the `Model` (and potentially related models). This leads to better search results with very little effort.

3.4.4 Method Reference

`__init__`

```
SearchField.__init__(self, model_attr=None, use_template=False, template_name=None,
                    document=False, indexed=True, stored=True, faceted=False,
                    default=NOT_PROVIDED, null=False, index_fieldname=None,
                    facet_class=None, boost=1.0, weight=None)
```

Instantiates a fresh `SearchField` instance.

`has_default`

```
SearchField.has_default(self)
```

Returns a boolean of whether this field has a default value.

`prepare`

```
SearchField.prepare(self, obj)
```

Takes data from the provided object and prepares it for storage in the index.

`prepare_template`

```
SearchField.prepare_template(self, obj)
```

Flattens an object for indexing.

This loads a template (`search/indexes/{app_label}/{model_name}_{field_name}.txt`) and returns the result of rendering that template. `object` will be in its context.

`convert`

```
SearchField.convert(self, value)
```

Handles conversion between the data found and the type of the field.

Extending classes should override this method and provide correct data coercion.

3.5 SearchResult API

class SearchResult (*app_label, model_name, pk, score, **kwargs*)

The `SearchResult` class provides structure to the results that come back from the search index. These objects are what a `SearchQuerySet` will return when evaluated.

3.5.1 Attribute Reference

The class exposes the following useful attributes/properties:

- `app_label` - The application the model is attached to.
- `model_name` - The model's name.
- `pk` - The primary key of the model.
- `score` - The score provided by the search engine.
- `object` - The actual model instance (lazy loaded).
- `model` - The model class.
- `verbose_name` - A prettier version of the model's class name for display.
- `verbose_name_plural` - A prettier version of the model's *plural* class name for display.
- `search_index` - Returns the `SearchIndex` class associated with this result.
- `distance` - On geo-spatial queries, this returns a `Distance` object representing the distance the result was from the focused point.

3.5.2 Method Reference

`content_type`

`SearchResult.content_type` (*self*)

Returns the content type for the result's model instance.

`get_additional_fields`

`SearchResult.get_additional_fields` (*self*)

Returns a dictionary of all of the fields from the raw result.

Useful for serializing results. Only returns what was seen from the search engine, so it may have extra fields Haystack's indexes aren't aware of.

`get_stored_fields`

`SearchResult.get_stored_fields` (*self*)

Returns a dictionary of all of the stored fields from the `SearchIndex`.

Useful for serializing results. Only returns the fields Haystack's indexes are aware of as being 'stored'.

3.6 SearchQuery API

class SearchQuery (*using=DEFAULT_ALIAS*)

The `SearchQuery` class acts as an intermediary between `SearchQuerySet`'s abstraction and `SearchBackend`'s actual search. Given the metadata provided by `SearchQuerySet`, `SearchQuery` build the actual query and interacts with the `SearchBackend` on `SearchQuerySet`'s behalf.

This class must be at least partially implemented on a per-backend basis, as portions are highly specific to the backend. It usually is bundled with the accompanying `SearchBackend`.

Most people will **NOT** have to use this class directly. `SearchQuerySet` handles all interactions with `SearchQuery` objects and provides a nicer interface to work with.

Should you need advanced/custom behavior, you can supply your version of `SearchQuery` that overrides/extends the class in the manner you see fit. You can either hook it up in a `BaseEngine` subclass or `SearchQuerySet` objects take a kwarg parameter `query` where you can pass in your class.

3.6.1 SQ Objects

For expressing more complex queries, especially involving AND/OR/NOT in different combinations, you should use SQ objects. Like `django.db.models.Q` objects, SQ objects can be passed to `SearchQuerySet.filter` and use the familiar unary operators (&, | and ~) to generate complex parts of the query.

Warning: Any data you pass to SQ objects is passed along **unescaped**. If you don't trust the data you're passing along, you should use the `clean` method on your `SearchQuery` to sanitize the data.

Example:

```
from haystack.query import SQ

# We want "title: Foo AND (tags:bar OR tags:moof)"
sqs = SearchQuerySet().filter(title='Foo').filter(SQ(tags='bar') | SQ(tags='moof'))

# To clean user-provided data:
sqs = SearchQuerySet()
clean_query = sqs.query.clean(user_query)
sqs = sqs.filter(SQ(title=clean_query) | SQ(tags=clean_query))
```

Internally, the `SearchQuery` object maintains a tree of SQ objects. Each SQ object supports what field it looks up against, what kind of lookup (i.e. the `__` filters), what value it's looking for, if it's a AND/OR/NOT and tracks any children it may have. The `SearchQuery.build_query` method starts with the root of the tree, building part of the final query at each node until the full final query is ready for the `SearchBackend`.

3.6.2 Backend-Specific Methods

When implementing a new backend, the following methods will need to be created:

build_query_fragment

`SearchQuery.build_query_fragment` (*self, field, filter_type, value*)

Generates a query fragment from a field, filter type and a value.

Must be implemented in backends as this will be highly backend specific.

3.6.3 Inheritable Methods

The following methods have a complete implementation in the base class and can largely be used unchanged.

build_query

`SearchQuery.build_query(self)`

Interprets the collected query metadata and builds the final query to be sent to the backend.

build_params

`SearchQuery.build_params(self, spelling_query=None)`

Generates a list of params to use when searching.

clean

`SearchQuery.clean(self, query_fragment)`

Provides a mechanism for sanitizing user input before presenting the value to the backend.

A basic (override-able) implementation is provided.

run

`SearchQuery.run(self, spelling_query=None, **kwargs)`

Builds and executes the query. Returns a list of search results.

Optionally passes along an alternate query for spelling suggestions.

Optionally passes along more kwargs for controlling the search query.

run_mlt

`SearchQuery.run_mlt(self, **kwargs)`

Executes the More Like This. Returns a list of search results similar to the provided document (and optionally query).

run_raw

`SearchQuery.run_raw(self, **kwargs)`

Executes a raw query. Returns a list of search results.

get_count

`SearchQuery.get_count(self)`

Returns the number of results the backend found for the query.

If the query has not been run, this will execute the query and store the results.

`get_results`

`SearchQuery.get_results(self, **kwargs)`

Returns the results received from the backend.

If the query has not been run, this will execute the query and store the results.

`get_facet_counts`

`SearchQuery.get_facet_counts(self)`

Returns the results received from the backend.

If the query has not been run, this will execute the query and store the results.

`boost_fragment`

`SearchQuery.boost_fragment(self, boost_word, boost_value)`

Generates query fragment for boosting a single word/value pair.

`matching_all_fragment`

`SearchQuery.matching_all_fragment(self)`

Generates the query that matches all documents.

`add_filter`

`SearchQuery.add_filter(self, expression, value, use_not=False, use_or=False)`

Narrows the search by requiring certain conditions.

`add_order_by`

`SearchQuery.add_order_by(self, field)`

Orders the search result by a field.

`clear_order_by`

`SearchQuery.clear_order_by(self)`

Clears out all ordering that has been already added, reverting the query to relevancy.

`add_model`

`SearchQuery.add_model(self, model)`

Restricts the query requiring matches in the given model.

This builds upon previous additions, so you can limit to multiple models by chaining this method several times.

set_limits

`SearchQuery.set_limits(self, low=None, high=None)`

Restricts the query by altering either the start, end or both offsets.

clear_limits

`SearchQuery.clear_limits(self)`

Clears any existing limits.

add_boost

`SearchQuery.add_boost(self, term, boost_value)`

Adds a boosted term and the amount to boost it to the query.

raw_search

`SearchQuery.raw_search(self, query_string, **kwargs)`

Runs a raw query (no parsing) against the backend.

This method causes the `SearchQuery` to ignore the standard query generating facilities, running only what was provided instead.

Note that any kwargs passed along will override anything provided to the rest of the `SearchQuerySet`.

more_like_this

`SearchQuery.more_like_this(self, model_instance)`

Allows backends with support for “More Like This” to return results similar to the provided instance.

add_stats_query

`SearchQuery.add_stats_query(self, stats_field, stats_facets)`

Adds stats and stats_facets queries for the Solr backend.

add_highlight

`SearchQuery.add_highlight(self)`

Adds highlighting to the search results.

add_within

`SearchQuery.add_within(self, field, point_1, point_2):`

Adds bounding box parameters to search query.

`add_dwithin`

`SearchQuery.add_dwithin(self, field, point, distance):`

Adds radius-based parameters to search query.

`add_distance`

`SearchQuery.add_distance(self, field, point):`

Denotes that results should include distance measurements from the point passed in.

`add_field_facet`

`SearchQuery.add_field_facet(self, field, **options)`

Adds a regular facet on a field.

`add_date_facet`

`SearchQuery.add_date_facet(self, field, start_date, end_date, gap_by, gap_amount)`

Adds a date-based facet on a field.

`add_query_facet`

`SearchQuery.add_query_facet(self, field, query)`

Adds a query facet on a field.

`add_narrow_query`

`SearchQuery.add_narrow_query(self, query)`

Narrows a search to a subset of all documents per the query.

Generally used in conjunction with faceting.

`set_result_class`

`SearchQuery.set_result_class(self, klass)`

Sets the result class to use for results.

Overrides any previous usages. If `None` is provided, Haystack will revert back to the default `SearchResult` object.

`using`

`SearchQuery.using(self, using=None)`

Allows for overriding which connection should be used. This disables the use of routers when performing the query.

If `None` is provided, it has no effect on what backend is used.

3.7 SearchBackend API

class SearchBackend (*connection_alias*, ***connection_options*)

The `SearchBackend` class handles interaction directly with the backend. The search query it performs is usually fed to it from a `SearchQuery` class that has been built for that backend.

This class must be at least partially implemented on a per-backend basis and is usually accompanied by a `SearchQuery` class within the same module.

Unless you are writing a new backend, it is unlikely you need to directly access this class.

3.7.1 Method Reference

update

`SearchBackend.update` (*self*, *index*, *iterable*)

Updates the backend when given a `SearchIndex` and a collection of documents.

This method **MUST** be implemented by each backend, as it will be highly specific to each one.

remove

`SearchBackend.remove` (*self*, *obj_or_string*)

Removes a document/object from the backend. Can be either a model instance or the identifier (i.e. `app_name.model_name.id`) in the event the object no longer exists.

This method **MUST** be implemented by each backend, as it will be highly specific to each one.

clear

`SearchBackend.clear` (*self*, *models=[]*)

Clears the backend of all documents/objects for a collection of models.

This method **MUST** be implemented by each backend, as it will be highly specific to each one.

search

`SearchBackend.search` (*self*, *query_string*, *sort_by=None*, *start_offset=0*, *end_offset=None*, *fields=''*, *highlight=False*, *facets=None*, *date_facets=None*, *query_facets=None*, *narrow_queries=None*, *spelling_query=None*, *limit_to_registered_models=None*, *result_class=None*, ***kwargs*)

Takes a query to search on and returns dictionary.

The query should be a string that is appropriate syntax for the backend.

The returned dictionary should contain the keys 'results' and 'hits'. The 'results' value should be an iterable of populated `SearchResult` objects. The 'hits' should be an integer count of the number of matched results the search backend found.

This method **MUST** be implemented by each backend, as it will be highly specific to each one.

`extract_file_contents`

`SearchBackend.extract_file_contents` (*self*, *file_obj*)

Perform text extraction on the provided file or file-like object. Returns either `None` or a dictionary containing the keys `contents` and `metadata`. The `contents` field will always contain the extracted text content returned by the underlying search engine but `metadata` may vary considerably based on the backend and the input file.

`prep_value`

`SearchBackend.prep_value` (*self*, *value*)

Hook to give the backend a chance to prep an attribute value before sending it to the search engine.

By default, just force it to unicode.

`more_like_this`

`SearchBackend.more_like_this` (*self*, *model_instance*, *additional_query_string=None*, *result_class=None*)

Takes a model object and returns results the backend thinks are similar.

This method **MUST** be implemented by each backend, as it will be highly specific to each one.

`build_schema`

`SearchBackend.build_schema` (*self*, *fields*)

Takes a dictionary of fields and returns schema information.

This method **MUST** be implemented by each backend, as it will be highly specific to each one.

`build_models_list`

`SearchBackend.build_models_list` (*self*)

Builds a list of models for searching.

The `search` method should use this and the `django_ct` field to narrow the results (unless the user indicates not to). This helps ignore any results that are not currently handled models and ensures consistent caching.

3.8 Architecture Overview

3.8.1 SearchQuerySet

One main implementation.

- Standard API that loosely follows `QuerySet`
- Handles most queries
- Allows for custom “parsing”/building through API
- Dispatches to `SearchQuery` for actual query

- Handles automatically creating a query
- Allows for raw queries to be passed straight to backend.

3.8.2 SearchQuery

Implemented per-backend.

- Method for building the query out of the structured data.
- Method for cleaning a string of reserved characters used by the backend.

Main class provides:

- Methods to add filters/models/order-by/boost/limits to the search.
- Method to perform a raw search.
- Method to get the number of hits.
- Method to return the results provided by the backend (likely not a full list).

3.8.3 SearchBackend

Implemented per-backend.

- Connects to search engine
- Method for saving new docs to index
- Method for removing docs from index
- Method for performing the actual query

3.8.4 SearchSite

One main implementation.

- Standard API that loosely follows `django.contrib.admin.sites.AdminSite`
- Handles registering/unregistering models to search on a per-site basis.
- Provides a means of adding custom indexes to a model, like `ModelAdmins`.

3.8.5 SearchIndex

Implemented per-model you wish to index.

- Handles generating the document to be indexed.
- Populates additional fields to accompany the document.
- Provides a way to limit what types of objects get indexed.
- Provides a way to index the document(s).
- Provides a way to remove the document(s).

3.9 Backend Support

3.9.1 Supported Backends

- Solr
- Elasticsearch
- Whoosh
- Xapian

3.9.2 Backend Capabilities

Solr

Complete & included with Haystack.

- Full SearchQuerySet support
- Automatic query building
- “More Like This” functionality
- Term Boosting
- Faceting
- Stored (non-indexed) fields
- Highlighting
- Spatial search
- Requires: pysolr (2.0.13+) & Solr 3.5+

Elasticsearch

Complete & included with Haystack.

- Full SearchQuerySet support
- Automatic query building
- “More Like This” functionality
- Term Boosting
- Faceting (up to 100 facets)
- Stored (non-indexed) fields
- Highlighting
- Spatial search
- Requires: pyelasticsearch 0.4+ & Elasticsearch 0.17.7+

Whoosh

Complete & included with Haystack.

- Full SearchQuerySet support
- Automatic query building
- Term Boosting
- Stored (non-indexed) fields
- Highlighting
- Requires: whoosh (1.1.1+)

Xapian

Complete & available as a third-party download.

- Full SearchQuerySet support
- Automatic query building
- “More Like This” functionality
- Term Boosting
- Faceting
- Stored (non-indexed) fields
- Highlighting
- Requires: Xapian 1.0.5+ & python-xapian 1.0.5+
- Backend can be downloaded here: [xapian-haystack](#)

3.9.3 Backend Support Matrix

Back-end	SearchQuery-Set Support	Auto Query Building	More Like This	Term Boost	Faceting	Stored Fields	High-lighting	Spa-tial
Solr	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Elastic-search	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Whoosh	Yes	Yes	No	Yes	No	Yes	Yes	No
Xapian	Yes	Yes	Yes	Yes	Yes	Yes	Yes (plugin)	No

3.9.4 Wishlist

The following are search backends that would be nice to have in Haystack but are licensed in a way that prevents them from being officially bundled. If the community expresses interest in any of these, there may be future development.

- [Riak](#)
- [Lupyne](#)
- [Sphinx](#)

Sphinx

This backend is unlikely to be built. Sphinx is pretty gimpy & doesn't do blended search results across all models the way the other engines can. Very limited featureset as well.

- Full SearchQuerySet support
- Automatic query building
- Term Boosting
- Stored (non-indexed) fields
- Highlighting
- Requires: sphinxapi.py (Comes with Sphinx)

3.10 Haystack Settings

As a way to extend/change the default behavior within Haystack, there are several settings you can alter within your `settings.py`. This is a comprehensive list of the settings Haystack recognizes.

3.10.1 HAYSTACK_DEFAULT_OPERATOR

Optional

This setting controls what the default behavior for chaining SearchQuerySet filters together is.

Valid options are:

```
HAYSTACK_DEFAULT_OPERATOR = 'AND'
HAYSTACK_DEFAULT_OPERATOR = 'OR'
```

Defaults to AND.

3.10.2 HAYSTACK_CONNECTIONS

Required

This setting controls which backends should be available. It should be a dictionary of dictionaries resembling the following (complete) example:

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.solr_backend.SolrEngine',
        'URL': 'http://localhost:9001/solr/default',
        'TIMEOUT': 60 * 5,
        'INCLUDE_SPELLING': True,
        'BATCH_SIZE': 100,
        'EXCLUDED_INDEXES': ['thirdpartyapp.search_indexes.BarIndex'],
    },
    'autocomplete': {
        'ENGINE': 'haystack.backends.whoosh_backend.WhooshEngine',
        'PATH': '/home/search/whoosh_index',
        'STORAGE': 'file',
        'POST_LIMIT': 128 * 1024 * 1024,
        'INCLUDE_SPELLING': True,
```

```

    'BATCH_SIZE': 100,
    'EXCLUDED_INDEXES': ['thirdpartyapp.search_indexes.BarIndex'],
  },
  'slave': {
    'ENGINE': 'xapian_backend.XapianEngine',
    'PATH': '/home/search/xapian_index',
    'INCLUDE_SPELLING': True,
    'BATCH_SIZE': 100,
    'EXCLUDED_INDEXES': ['thirdpartyapp.search_indexes.BarIndex'],
  },
  'db': {
    'ENGINE': 'haystack.backends.simple_backend.SimpleEngine',
    'EXCLUDED_INDEXES': ['thirdpartyapp.search_indexes.BarIndex'],
  }
}

```

No default for this setting is provided.

The main keys (`default` & `friends`) are identifiers for your application. You can use them any place the API exposes using as a method or kwarg.

There must always be at least a `default` key within this setting.

The `ENGINE` option is required for all backends & should point to the `BaseEngine` subclass for the backend.

Additionally, each backend may have additional options it requires:

- Solr
 - URL - The URL to the Solr core.
- Whoosh
 - PATH - The filesystem path to where the index data is located.
- Xapian
 - PATH - The filesystem path to where the index data is located.

The following options are optional:

- `INCLUDE_SPELLING` - Include spelling suggestions. Default is `False`
- `BATCH_SIZE` - How many records should be updated at once via the management commands. Default is 1000.
- `TIMEOUT` - (Solr-only) How long to wait (in seconds) before the connection times out. Default is 10.
- `STORAGE` - (Whoosh-only) Which storage engine to use. Accepts `file` or `ram`. Default is `file`.
- `POST_LIMIT` - (Whoosh-only) How large the file sizes can be. Default is `128 * 1024 * 1024`.
- `FLAGS` - (Xapian-only) A list of flags to use when querying the index.
- `EXCLUDED_INDEXES` - A list of strings (as Python import paths) to indexes you do **NOT** want included. Useful for omitting third-party things you don't want indexed or for when you want to replace an index.

3.10.3 HAYSTACK_ROUTERS

Optional

This setting controls how routing is performed to allow different backends to handle updates/deletes/reads.

An example:

```
HAYSTACK_ROUTERS = ['search_routers.MasterSlaveRouter', 'haystack.routers.DefaultRouter']
```

Defaults to `['haystack.routers.DefaultRouter']`.

3.10.4 HAYSTACK_SIGNAL_PROCESSOR

Optional

This setting controls what `SignalProcessor` class is used to handle Django's signals & keep the search index up-to-date.

An example:

```
HAYSTACK_SIGNAL_PROCESSOR = 'haystack.signals.RealtimeSignalProcessor'
```

Defaults to `'haystack.signals.BaseSignalProcessor'`.

3.10.5 HAYSTACK_DOCUMENT_FIELD

Optional

This setting controls what fieldname Haystack relies on as the default field for searching within.

An example:

```
HAYSTACK_DOCUMENT_FIELD = 'wall_o_text'
```

Defaults to `text`.

3.10.6 HAYSTACK_SEARCH_RESULTS_PER_PAGE

Optional

This setting controls how many results are shown per page when using the included `SearchView` and its subclasses.

An example:

```
HAYSTACK_SEARCH_RESULTS_PER_PAGE = 50
```

Defaults to `20`.

3.10.7 HAYSTACK_CUSTOM_HIGHLIGHTER

Optional

This setting allows you to specify your own custom `Highlighter` implementation for use with the `{% highlight %}` template tag. It should be the full path to the class.

An example:

```
HAYSTACK_CUSTOM_HIGHLIGHTER = 'myapp.utils.BorkHighlighter'
```

No default is provided. Haystack automatically falls back to the default implementation.

3.10.8 HAYSTACK_ITERATOR_LOAD_PER_QUERY

Optional

This setting controls the number of results that are pulled at once when iterating through a `SearchQuerySet`. If you generally consume large portions at a time, you can bump this up for better performance.

Note: This is not used in the case of a slice on a `SearchQuerySet`, which already overrides the number of results pulled at once.

An example:

```
HAYSTACK_ITERATOR_LOAD_PER_QUERY = 100
```

The default is 10 results at a time.

3.10.9 HAYSTACK_LIMIT_TO_REGISTERED_MODELS

Optional

This setting allows you to control whether or not Haystack will limit the search results seen to just the models registered. It should be a boolean.

If your search index is never used for anything other than the models registered with Haystack, you can turn this off and get a small to moderate performance boost.

An example:

```
HAYSTACK_LIMIT_TO_REGISTERED_MODELS = False
```

Default is `True`.

3.10.10 HAYSTACK_ID_FIELD

Optional

This setting allows you to control what the unique field name used internally by Haystack is called. Rarely needed unless your field names collide with Haystack's defaults.

An example:

```
HAYSTACK_ID_FIELD = 'my_id'
```

Default is `id`.

3.10.11 HAYSTACK_DJANGO_CT_FIELD

Optional

This setting allows you to control what the content type field name used internally by Haystack is called. Rarely needed unless your field names collide with Haystack's defaults.

An example:

```
HAYSTACK_DJANGO_CT_FIELD = 'my_django_ct'
```

Default is `django_ct`.

3.10.12 HAYSTACK_DJANGO_ID_FIELD

Optional

This setting allows you to control what the primary key field name used internally by Haystack is called. Rarely needed unless your field names collide with Haystack's defaults.

An example:

```
HAYSTACK_DJANGO_ID_FIELD = 'my_django_id'
```

Default is `django_id`.

3.11 Utilities

Included here are some of the general use bits included with Haystack.

3.11.1 `get_identifier`

`get_identifier` (*obj_or_string*)

Get an unique identifier for the object or a string representing the object.

If not overridden, uses `<app_label>.<object_name>.<pk>`.

Developing

Finally, if you're looking to help out with the development of Haystack, the following links should help guide you on running tests and creating additional backends:

4.1 Running Tests

4.1.1 Dependencies

Everything

The simplest way to get up and running with Haystack's tests is to run:

```
pip install -r tests/requirements.txt
```

This installs all of the backend libraries & all dependencies for getting the tests going. You will still have to setup search servers (for running Solr tests, the spatial Solr tests & the Elasticsearch tests).

Cherry-Picked

If you'd rather not run all the tests, install only the backends you need. Additionally, Haystack uses the [Mock](#) library for testing. You will need to install it before running the tests:

```
pip install mock
```

4.1.2 Core Haystack Functionality

In order to test Haystack with the minimum amount of unnecessary mocking and to stay as close to real-world use as possible, Haystack ships with a test app (called `core`) within the `django-haystack/tests` directory.

In the event you need to run Haystack's tests (such as testing bugfixes/modifications), here are the steps to getting them running:

```
cd django-haystack/tests
export PYTHONPATH=`pwd`/..:`pwd`
django-admin.py test core --settings=settings
```

Haystack is maintained with all tests passing at all times, so if you receive any errors during testing, please check your setup and file a report if the errors persist.

4.1.3 Backends

If you want to test a backend, the steps are the same with the exception of the settings module and the app to test. To test an engine, use the `engine_settings` module within the `tests` directory, substituting the engine for the name of the proper backend. You'll also need to specify the app for that engine. For instance, to run the Solr backend's tests:

```
cd django-haystack/tests
export PYTHONPATH=`pwd`/..:`pwd`
django-admin.py test solr_tests --settings=solr_settings
```

Or, to run the Whoosh backend's tests:

```
cd django-haystack/tests
export PYTHONPATH=`pwd`/..:`pwd`
django-admin.py test whoosh_tests --settings=whoosh_settings
```

Or, to run the Elasticsearch backend's tests:

```
cd django-haystack/tests
export PYTHONPATH=`pwd`/..:`pwd`
django-admin.py test elasticsearch_tests --settings=elasticsearch_settings
```

Configuring Solr

Haystack assumes that you have a Solr server running on port 9001 which uses the schema and configuration provided in the `tests/` directory. Currently, these steps will result in a working test server:

1. Download the current Solr release from <http://lucene.apache.org/solr/>
2. Copy `tests/solrconfig.xml` to `example/solr/conf/solrconfig.xml`
3. Copy `tests/solr_test_schema.xml` to `example/solr/conf/schema.xml`
4. Change into the `example` directory
5. Start Solr: `java -Djetty.port=9001 -jar start.jar`

4.2 Creating New Backends

The process should be fairly simple.

1. Create new backend file. Name is important.
2. Two classes inside.
 - (a) `SearchBackend` (inherit from `haystack.backends.BaseSearchBackend`)
 - (b) `SearchQuery` (inherit from `haystack.backends.BaseSearchQuery`)

4.2.1 SearchBackend

Responsible for the actual connection and low-level details of interacting with the backend.

- Connects to search engine
- Method for saving new docs to index

- Method for removing docs from index
- Method for performing the actual query

4.2.2 SearchQuery

Responsible for taking structured data about the query and converting it into a backend appropriate format.

- Method for creating the backend specific query - `build_query`.

Requirements

Haystack has a relatively easily-met set of requirements.

- Python 2.5+
- Django 1.3+

Additionally, each backend has its own requirements. You should refer to [Installing Search Engines](#) for more details.

Commercial Support

If you're using Haystack in a commercial environment, paid support is available from [Toast Driven](#). Services offered include:

- Advice/help with setup
- Implementation in your project
- Bugfixes in Haystack itself
- Features in Haystack itself

If you're interested, please contact Daniel Lindsley (daniel@toastdriven.com).

Symbols

`__init__()` (SearchField method), 94

A

`add_boost()` (SearchQuery method), 99
`add_date_facet()` (SearchQuery method), 100
`add_field_facet()` (SearchQuery method), 100
`add_filter()` (SearchQuery method), 98
`add_highlight()` (SearchQuery method), 99
`add_model()` (SearchQuery method), 98
`add_narrow_query()` (SearchQuery method), 100
`add_order_by()` (SearchQuery method), 98
`add_query_facet()` (SearchQuery method), 100
`add_stats_query()` (SearchQuery method), 99
`auto_query()` (SearchQuerySet method), 72

B

`best_match()` (SearchQuerySet method), 74
`boost()` (SearchQuerySet method), 69
`boost_fragment()` (SearchQuery method), 98
`build_models_list()` (SearchBackend method), 102
`build_params()` (SearchQuery method), 97
`build_query()` (SearchQuery method), 97
`build_query_fragment()` (SearchQuery method), 96
`build_queryset()` (SearchIndex method), 85
`build_schema()` (SearchBackend method), 102

C

`clean()` (SearchQuery method), 97
`clear()` (SearchBackend method), 101
`clear()` (SearchIndex method), 86
`clear_limits()` (SearchQuery method), 99
`clear_order_by()` (SearchQuery method), 98
`content_type()` (SearchResult method), 95
`convert()` (SearchField method), 94
`count()` (SearchQuerySet method), 74

D

`date_facet()` (SearchQuerySet method), 69
`default` (SearchField attribute), 92

`distance()` (SearchQuerySet method), 63
`document` (SearchField attribute), 92
`dwithin()` (SearchQuerySet method), 62

E

`exclude()` (SearchQuerySet method), 67
`extract_file_contents()` (SearchBackend method), 102

F

`facet()` (SearchQuerySet method), 69
`facet_counts()` (SearchQuerySet method), 74
`filter()` (SearchQuerySet method), 67
`filter_and()` (SearchQuerySet method), 68
`filter_or()` (SearchQuerySet method), 68

G

`get_additional_fields()` (SearchResult method), 95
`get_content_field()` (SearchIndex method), 85
`get_count()` (SearchQuery method), 97
`get_facet_counts()` (SearchQuery method), 98
`get_identifier()` (built-in function), 110
`get_model()` (SearchIndex method), 84
`get_results()` (SearchQuery method), 98
`get_stored_fields()` (SearchResult method), 95
`get_updated_field()` (SearchIndex method), 86

H

`has_default()` (SearchField method), 94
`haystack.inputs.AltParser` (built-in class), 90
`haystack.inputs.AutoQuery` (built-in class), 89
`haystack.inputs.Clean` (built-in class), 88
`haystack.inputs.Exact` (built-in class), 89
`haystack.inputs.Not` (built-in class), 89
`haystack.inputs.Raw` (built-in class), 88
`highlight()` (SearchQuerySet method), 68

I

`index_fieldname` (SearchField attribute), 92
`index_queryset()` (SearchIndex method), 84
`indexed` (SearchField attribute), 92

L

latest() (SearchQuerySet method), 74
load_all() (SearchQuerySet method), 72
load_all_queryset() (RelatedSearchQuerySet method), 78
load_all_queryset() (SearchIndex method), 86
load_all_queryset() (SearchQuerySet method), 72

M

matching_all_fragment() (SearchQuery method), 98
model_attr (SearchField attribute), 93
models() (SearchQuerySet method), 68
more_like_this() (SearchBackend method), 102
more_like_this() (SearchQuery method), 99
more_like_this() (SearchQuerySet method), 73

N

narrow() (SearchQuerySet method), 71
null (SearchField attribute), 93

O

order_by() (SearchQuerySet method), 68

P

prep_value() (SearchBackend method), 102
prepare() (SearchField method), 94
prepare() (SearchIndex method), 85
prepare_template() (SearchField method), 94

Q

query_facet() (SearchQuerySet method), 70

R

raw_search() (SearchQuery method), 99
raw_search() (SearchQuerySet method), 71
read_queryset() (SearchIndex method), 84
reindex() (SearchIndex method), 86
remove() (SearchBackend method), 101
remove_object() (SearchIndex method), 85
result_class() (SearchQuerySet method), 69
run() (SearchQuery method), 97
run_mlt() (SearchQuery method), 97
run_raw() (SearchQuery method), 97

S

search() (SearchBackend method), 101
SearchBackend (built-in class), 101
SearchField (built-in class), 90
SearchIndex (built-in class), 79
SearchQuery (built-in class), 96
SearchQuerySet (built-in class), 65
SearchResult (built-in class), 95
set_limits() (SearchQuery method), 99
set_result_class() (SearchQuery method), 100

should_update() (SearchIndex method), 86
spelling_suggestion() (SearchQuerySet method), 76
stored (SearchField attribute), 93

T

template_name (SearchField attribute), 93

U

update() (SearchBackend method), 101
update() (SearchIndex method), 85
update_object() (SearchIndex method), 85
use_template (SearchField attribute), 94
using() (SearchQuery method), 100
using() (SearchQuerySet method), 73

V

values() (SearchQuerySet method), 76
values_list() (SearchQuerySet method), 76

W

within() (SearchQuerySet method), 61