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

**Note:** This documentation represents the 1.0.X version of Haystack. For other versions of the documentation: dev.
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 app 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.

This tutorial assumes that you have a basic familiarity with the various major parts of Django (models/forms/views/settings/URLConf's) 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`:

```python
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`).

**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_SITECONF` is a required settings and should provide a Python import path to a file where you keep your `SearchSite` configurations in. This will be explained in the next step, but for now, add the following settings (substituting your correct information) and create an empty file at that path:

```python
HAYSTACK_SITECONF = 'myproject.search_sites'
```

`HAYSTACK_SEARCH_ENGINE` is a required setting and should be one of the following:

- `solr`
- `whoosh`
- `xapian` *(if you installed `xapian-haystack`)*
- `dummy`

Example:

```python
HAYSTACK_SEARCH_ENGINE = 'whoosh'
```

Additionally, backends may require additional information.

**Solr**

Requires setting `HAYSTACK_SOLR_URL` to be the URL where your Solr is running at.

Example:

```python
HAYSTACK_SOLR_URL = 'http://127.0.0.1:8983/solr'
# ...or for multicore...
HAYSTACK_SOLR_URL = 'http://127.0.0.1:8983/solr/mysite'
```

**Whoosh**

Requires setting `HAYSTACK_WHOOSH_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:
Xapian

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

Requires setting HAYSTACK_XAPIAN_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:

HAYSTACK_XAPIAN_PATH = '/home/xapian/mysite_index'

Create A SearchSite

Within the empty file you created corresponding to your HAYSTACK_SITECONF, add the following code:

```python
import haystack
haystack.autodiscover()
```

This will create a default SearchSite instance, search through all of your INSTALLED_APPS for search_indexes.py and register all SearchIndex classes with the default SearchSite.

Note: You can configure more than one SearchSite as well as manually registering/unregistering indexes with them. However, these are rarely done in practice and are available for advanced use.

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 use a SearchIndex, you need to register it with the Model it applies to and the SearchSite it ought to belong to. Registering indexes in Haystack is very similar to the way you register models and ModelAdmin classes with the Django admin site.

To build a SearchIndex, all that’s necessary is to subclass SearchIndex, define the fields you want to store data with and register it.

We’ll create the following NoteSearchIndex 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.autodiscover() to automatically pick it up. The NoteSearchIndex should look like:

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

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

    def get_queryset(self):
        return Note.objects.filter(pub_date__lte=datetime.datetime.now())

site.register(Note, NoteIndex)

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 at search/indexes/myapp/note_text.txt and place the following inside:

```text
{{ 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 get-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:

```python
(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 getting (your template/block names will likely differ):
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.

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 `RealTimeSearchIndex` 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:

```python
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()

        # 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.
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:

```python
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')

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

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)**

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

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.

```python
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.

```python
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.

```python
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.

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

```python
class FacetedSearchView(SearchView):
    def __name__(self):
        return "FacetedSearchView"

    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 n] %}

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 its 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. It accepts no arguments. Use with caution.

This is an INTERACTIVE command and assumes that you do NOT wish to delete the entire index.
1.5.2 update_index

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.
'--batch-size':
   Number of items to index at once. Default is 1000.
'--site':
   The site object to use when reindexing (like 'search_sites.mysite').
```

Using --verbosity=2 with this command shows individual batches being sent, which is useful when debugging.

Note: This command ONLY updates records in the index. It does NOT handle deletions, so you may need to write a separate script that handles deleted models, such as a queue consumer or something that runs through all records and tries to load the model for it. Alternatively, you can use the RealTimeSearchIndex, which will automatically handle deletions.

1.5.3 rebuild_index

A shortcut for clear_index followed by update_index. It accepts no arguments. 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 no arguments.

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
1.7.4 Eldarion

Various projects.
Using: Solr
  - http://eldarion.com/

1.7.5 Sunlight Labs

For general search.
Using: Whoosh & Solr
  - http://sunlightlabs.com/
  - http://subsidyscope.com/

1.7.6 NASA

For general search on an internal site called SMD Spacebook 1.1.
Using: Solr

1.7.7 AllForLocal

For general search.
  - http://www.allforlocal.com/

1.7.8 HUGE

Various projects.
Using: Solr
  - http://hugeinc.com/
  - http://houselogic.com/

1.7.9 Brick Design

For search on Explore.
Using: Solr
  - http://bricksf.com/
  - http://explore.org/
1.8 Installing Search Engines

1.8.1 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 requires Solr 1.3+. Installation is relatively simple:

```
curl -O http://apache.mirrors.tds.net/lucene/solr/1.3.0/apache-solr-1.3.0.tgz
tar xvzf apache-solr-1.3.0.tgz
cd apache-solr-1.3.0
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-1.3.0/example/solr/conf/schema.xml`. Then restart Solr.

You’ll also need a Solr binding, `pysolr`. The official `pysolr` package, distributed via PyPI, is the best version to use (2.0.9+). Place `pysolr.py` somewhere on your `PYTHONPATH`.

Note: `pysolr` has its 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 setup the `MoreLikeThisHandler`. Add 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="default">default</str>
    <str name="field">text</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">text</str> portion will be specific to your SearchIndex classes (in this case, assuming the main field is called text). This should be the same as the <defaultSearchField> in your schema.xml.

1.8.2 Whoosh

Official Download Location: http://whoosh.ca/

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. With the 0.3.1+ releases, Whoosh has become much more performant, stable and better tested. The current recommended version is 0.3.18. You can install via PyPI using:

```
sudo easy_install whoosh
# ... or ...
sudo pip install whoosh==0.3.18
```

Note that, while capable otherwise, the Whoosh backend does not currently support “More Like This” or faceting.

1.8.3 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 supported backend but is not included in Haystack proper due to licensing. 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.

1.8. Installing Search Engines
1.9 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.

1.9.1 “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.)

1.9.2 “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_sites.py` that runs haystack.autodiscover?
- Have you registered your models with the main haystack.site (usually within your `search_indexes.py`)?
- Do you have data in your database?
- Have you run a `.manage.py rebuild_index` to index all of your content?
- Start a Django shell (`.manage.py shell`) and try:
  ```python
  >>> from haystack.query import SearchQuerySet
  >>> sqs = SearchQuerySet().all()
  >>> sqs.count()
  ```
  You should get back an integer > 0. If not, check the above and reindex.
  ```python
  >>> 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.
1.9.3 “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 is locks the index files for writing while another process/thread tries to access them. This is a common error when using RealTimeSearchIndex with Whoosh under any kind of load, which is why it’s only recommended for small sites or development.

A way to solve this is to subclass SearchIndex instead:

```python
from haystack import indexes

# Change from:
#
# class MySearchIndex(indexes.RealTimeSearchIndex):
#
# to:
class MySearchIndex(indexes.SearchIndex):
  ...
```

The final step 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.

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.
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 been 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 %} template tag 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 one query for each result, up from a single query for each type of model with `load_all=True`.

### 2.1.3 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. By using the `RealTimeSearchIndex` class instead of the `SearchIndex` class, Haystack will automatically update the index whenever a model is saved/deleted.

You can find more information within the `SearchIndex API` documentation.

### 2.1.4 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 `QueuedSearchIndex` class and creating a queue processing script to handle the actual updates.

For the `QueuedSearchIndex`, simply inherit from the `SearchIndex` provided by Haystack and override the `_setup_save/_setup_delete` methods. These methods usually attach themselves to their model’s `post_save/post_delete` signals and call the backend to update or remove a record. You should override this behavior and place a message in your queue of choice. At a minimum, you’ll want to include the model you’re indexing and the id of the model within that message, so that you can retrieve the proper index from the `SearchSite` in your consumer. Then alter all of your `SearchIndex` classes to inherit from this new class. Now all saves/deletes will be handled by the queue and you should receive a speed boost.

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:

```python
>>> 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> in real life.'
```

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

```python
>>> 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:

```python
# 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:

```python
>>> 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 rests, 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.

If the data in the fields you’re faceting on is complex, you may want to consider duplicating fields you’ll be faceting on and marking them as `indexed=False`. So to modify our existing example:

```python
class NoteIndex(indexes.SearchIndex):
    text = indexes.CharField(document=True, use_template=True)
    author = indexes.CharField(model_attr='user')
    author_exact = indexes.CharField(model_attr='user', indexed=False)
    pub_date = indexes.DateTimeField(model_attr='pub_date')
```

This informs the backend that no post-processing is to be done on the data (such as lower-case/stemming/tokenizing/etc.), allowing the faceting to be more exact. Duplication is suggested so that those fields are still searchable in the standard ways, so if all you’re using a given field for faceting, no duplication is needed.

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:

```python
>>> from haystack.query import SearchQuerySet
>>> sqs = SearchQuerySet().facet('author_exact')
>>> sqs.facet_counts()
{...
  'dates': {},
```

2.3. Faceting
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_exact field, 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:

```python
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_exact')

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

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

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.
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** *(site=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:

```python
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]
```
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:

```python
from haystack.query import SearchQuerySet
results = SearchQuerySet().exclude(content='hello').filter(content='world').order_by('-pub_date').boost('title', 0.5)[10:20]
```

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:

```python
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

```python
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.

```python
none
```

SearchQuerySet.none(self):

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

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

Narrows 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).

If a string with one or more spaces in it is specified as the value, an exact match will be performed on that phrase.

**Warning:** Any data you pass to filter is passed along unescaped. If you don’t trust the data you’re passing along, you should either use auto_query or use the clean method on your SearchQuery to sanitize the data.

Example:

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

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

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

# To escape user data:
sqs = SearchQuerySet()
sqs = sqs.filter(title=sqs.query.clean(user_query))

**exclude**

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

Narrows the search by ensuring certain attributes are not included.

**Warning:** Any data you pass to exclude is passed along unescaped. If you don’t trust the data you’re passing along, you should either use auto_query or use the clean method on your SearchQuery to sanitize the data.

Example:

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:

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

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

```python
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.

**Warning:** Whoosh only If you’re planning on ordering by an `IntegerField` using Whoosh, you’ll need to adequately zero-pad your numbers in the `prepare_FOO` method. This is because Whoosh uses UTF-8 string for everything, and from the schema, there is no way to know how a field should be treated.

**highlight**

`SearchQuerySet.highlight(self)`

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

```python
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>Foo</em>!"'
```

**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:

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

**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)

Adds faceting to a query for the provided field. You provide the field (from one of the SearchIndex classes) you like to facet on.

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), gap_by='day')

**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*')
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:

```python
# 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))
```

Please note that this 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.

**Warning:** Unlike many of the other methods on SearchQuerySet, this method does not chain by default (depends on the backend). Any other attributes on the SearchQuerySet are ignored and only the provided query is run.

Example:

```python
# In the case of Solr...
SearchQuerySet().raw_search('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:

```python
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)

Performs a best guess constructing the search query.

This method is intended for common use directly with a user’s query. It is a shortcut to the other API methods that
follows generally established search syntax without requiring each developer to implement their own parser.

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:
SearchQuerySet().auto_query('goldfish "old one eye" -tank')

# ... is identical to...
SearchQuerySet().filter(content='old one eye').filter(content='goldfish').exclude(content='tank')

This method is somewhat naive but works well enough for simple, common cases.

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"

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': {}
# }

spelling_suggestion

SearchQuerySet.spelling_suggestion(self, preferred_query=None)

Returns the spelling suggestion found by the query.

To work, you must set settings.HAYSTACK_INCLUDE_SPELLING (see Haystack Settings) 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 examples')
sqs.spelling_suggestion()  # u'more examples'

# ...or...
suggestion = SearchQuerySet().spelling_suggestion('moar examples')
suggestion  # u'more examples'

Field Lookups

The following lookup types are supported:

- exact
- gt
- gte
- lt
- lte
- in
• `startswith`

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.

Example:

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

# Identical to:

```python
SearchQuerySet().filter(content__exact='foo')
```

# Other usages look like:

```python
SearchQuerySet().filter(pub_date__gte=datetime.date(2008, 1, 1), pub_date__lt=datetime.date(2009, 1), author__in=['daniel', 'john', 'jane'])
```

### 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(model, backend=None)

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:

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

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

#### 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).

**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`:

```python
class NoteIndex(indexes.SearchIndex):
    text = indexes.CharField(document=True, use_template=True)
    author = indexes.CharField(model_attr='user')
    pub_date = indexes.DateTimeField(model_attr='pub_date')
    # Define the additional field.
    rendered = indexes.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:

```latex
{% for result in page.object_list %}
  <div class="search_result">
    {{ result.rendered }}
  </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 `RealTimeSearchIndex`, 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 is the only backend 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 `QueueSearchIndex` that, much like `RealTimeSearchIndex`, 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 `QueueSearchIndex` 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 in that you simply extend the same four methods as `RealTimeSearchIndex` and simply add messages to the queue of choice.

### 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:

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

    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:

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

    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.

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

    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:

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

    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:

```python
from haystack.indexes import CharField

class GeoPointField(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 analogous to Django’s Field.clean methods.

3.2. SearchIndex API
3.2.5 Search Index

get_queryset

SearchIndex.get_queryset(self)
Get the default QuerySet to index when doing a full update.
Subclasses can override this method to avoid indexing certain objects.

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)
Update the entire index.

update_object

SearchIndex.update_object(self, instance, **kwargs)
Update the index for a single object. Attached to the class’s post-save hook.

remove_object

SearchIndex.remove_object(self, instance, **kwargs)
Remove an object from the index. Attached to the class’s post-delete hook.

clear

SearchIndex.clear(self)
Clear the entire index.

reindex

SearchIndex.reindex(self)
Completely clear the index for this model and rebuild it.
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.
By default, returns all() on the model’s default manager.
Example:

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

    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.

3.2.6 RealTimeSearchIndex

The RealTimeSearchIndex provides all the same functionality as the standard SearchIndex. However, in addition, it connects to the post_save/post_delete signals of the model it’s registered with.
This means that anytime a model is saved or deleted, it’s automatically and immediately updated in the search index, yielding real-time search.

**Warning:** Not all backends deal well with the kind of document churn that can result from using the `RealTimeSearchIndex`. Solr is the only one that handles it gracefully. Additionally, this will add more overhead in terms of CPU usage, so you should be sure to accommodate for this and should have appropriate monitoring in place.

### 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:

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

# All Fields
class AllNoteIndex(indexes.ModelSearchIndex):
    class Meta:
        pass

# Blacklisted Fields
class LimitedNoteIndex(indexes.ModelSearchIndex):
    class Meta:
        excludes = ['user']

# Whitelisted Fields
class NoteIndex(indexes.ModelSearchIndex):
    class Meta:
        fields = ['user', 'pub_date']

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

48 Chapter 3. Reference
3.3 SearchField API

class SearchField

The SearchField and its subclasses provide 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.

3.3.1 Subclasses

Included with Haystack are the following field types:

- BooleanField
- CharField
- DateField
- DateTimeField
- FloatField
- IntegerField
- MultiValueField

3.3.2 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:

```python
from haystack import indexes

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

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.
3.3.3 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 data from this field will be searchable within the index. Default is True.

The companion of this option is `stored`.

**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:

```python
author = indexes.CharField(model_attr='last_name')
```

It also handles callables:

```python
author = indexes.CharField(model_attr='get_full_name')
```

And can look through relations:

```python
biography = indexes.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:

```python
bio = indexes.CharField(use_template=True, template_name='myapp/data/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.

### 3.3.4 Method Reference

#### __init__

`SearchField.__init__(self, model_attr=None, use_template=False, template_name=None, document=False, indexed=True, stored=True, default=NOT_PROVIDED, null=False)`

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.

3.4 SearchResult API

class SearchResult (app_label, model_name, pk, score, **kwags)
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.

3.4.1 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.

3.4.2 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’.

3.5 SearchSite API

class SearchSite
The SearchSite provides a way to collect the SearchIndexes that are relevant to the current site, much like ModelAdmins in the admin app.
This allows you to register indexes on models you don’t control (reusable apps, django.contrib, etc.) as well as customize on a per-site basis what indexes should be available (different indexes for different sites, same codebase).
A SearchSite instance(s) should be configured within a configuration file, which gets specified in your settings file as HAYSTACK_SITECONF. An example of this setting might be myproject.search_sites.

Warning: For a long time before the 1.0 release of Haystack, the convention was to place this configuration within your URLconf. This is no longer recommended as it can cause issues in certain production setups (Django 1.1+/mod_wsgi for example).

3.5.1 Autodiscovery
Since the common use case is to simply grab everything that is indexed for search, there is an autodiscovery mechanism which will pull in and register all indexes it finds within your project. To enable this, place the following code inside the file you specified as your HAYSTACK_SITECONF:

import haystack
haystack.autodiscover()

This will fully flesh-out the default SearchSite (at haystack.sites.site) for use. Since this site is used by default throughout Haystack, very little (if any) additional configuration will be needed.

3.5.2 Usage
If you need to narrow the indexes that get registered, you will need to manipulate a SearchSite. There are two ways to go about this, via either register or unregister.
If you want most of the indexes but want to forgo a specific one(s), you can setup the main site via autodiscover then simply unregister the one(s) you don’t want.
import haystack
haystack.autodiscover()

# Unregister the Rating index.
from ratings.models import Rating
haystack.sites.site.unregister(Rating)

Alternatively, you can manually register only the indexes you want:

from haystack import site
from ratings.models import Rating
from ratings.search_indexes import RatingIndex

site.register(Rating, RatingIndex)

### 3.5.3 Method Reference

**register**

SearchSite.register(self, model, index_class=None)

Registers a model with the site.
The model should be a Model class, not instances.
If no custom index is provided, a generic SearchIndex will be applied to the model.

**unregister**

SearchSite.unregister(self, model)

Unregisters a model’s corresponding index from the site.

**get_index**

SearchSite.get_index(self, model)

Provides the index that’s registered for a particular model.

**get_indexes**

SearchSite.get_indexes(self)

Provides a dictionary of all indexes that’re being used.

**get_indexed_models**

SearchSite.get_indexed_models(self)

Provides a list of all models being indexed.
all_searchfields

SearchSite.all_searchfields(self)

Builds a dictionary of all fields appearing in any of the SearchIndex instances registered with a site.

This is useful when building a schema for an engine. A dictionary is returned, with each key being a fieldname and the value being the SearchField class assigned to it.

update_object

SearchSite.update_object(self, instance)

Updates the instance’s data in the index.

A shortcut for updating on the instance’s index. Errors from get_index and update_object will be allowed to propagate.

remove_object

SearchSite.remove_object(self, instance)

Removes the instance’s data in the index.

A shortcut for removing on the instance’s index. Errors from get_index and remove_object will be allowed to propagate.

3.6 SearchQuery API

class SearchQuery(backend=None)

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. 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)
Builds and executes the query. Returns a list of search results.
Optionally passes along an alternate query for spelling suggestions.

run_mlt

SearchQuery.run_mlt(self)
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)
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)
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_highlight

SearchQuery.add_highlight(self)
Adds highlighting to the search results.

add_field_facet

SearchQuery.add_field_facet(self, field)
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)
Adds a existing facet on a field.

3.7 SearchBackend API

class SearchBackend(site=None)
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=True, **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.

**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)`

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_registered_models_list**

`SearchBackend.build_registered_models_list(self)`

Builds a list of registered 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 registered 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
- Lucene
- Whoosh

#### 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
- Requires: pysolr (2.0.9+) + Solr 1.3+

**Lucene**

- Full SearchQuerySet support
- Automatic query building
- Term Boosting
- Stored (non-indexed) fields
- Highlighting
- Requires: pylucene

**Whoosh**

Complete & included with Haystack.

- Full SearchQuerySet support
- Automatic query building
- Term Boosting
- Stored (non-indexed) fields
- Highlighting
- Requires: whoosh (0.3.15 - 0.3.18 - NOT the 1.X releases)

<table>
<thead>
<tr>
<th>Backend</th>
<th>SearchQuerySet Support</th>
<th>Auto Query Building</th>
<th>More Like This</th>
<th>Term Boost</th>
<th>Faceting</th>
<th>Stored Fields</th>
<th>Highlighting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solr</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Lucene</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Whoosh</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**3.9.3 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.

- Xapian
- Sphinx
- Hyper Estraier
Xapian

Complete but not included with Haystack.

- Full SearchQuerySet support
- Automatic query building
- Term Boosting
- “More Like This” functionality
- Faceting
- Stored (non-indexed) fields
- Highlighting
- Requires: xapian bindings included with Xapian

Sphinx

- Full SearchQuerySet support
- Automatic query building
- Term Boosting
- Stored (non-indexed) fields
- Highlighting
- Requires: sphinxapi.py (Comes with Sphinx)

Hyper Estraier

- Full SearchQuerySet support
- Automatic query building
- “More Like This” functionality
- Highlighting
- Requires: SWIG bindings

<table>
<thead>
<tr>
<th>Backend</th>
<th>SearchQuerySet Support</th>
<th>Auto Query Building</th>
<th>More Like This</th>
<th>Term Boost</th>
<th>Faceting</th>
<th>Stored Fields</th>
<th>Highlighting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xapian</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Sphinx</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes (plugin)</td>
</tr>
<tr>
<td>Hyper Estraier</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes (plugin)</td>
</tr>
</tbody>
</table>

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_SITECONF

Required
This setting controls what module should be loaded to setup your SearchSite. The module should be on your PYTHONPATH and should contain only the calls necessary to setup Haystack to your needs.

The convention is to name this file search_sites and place it in the same directory as your settings.py and/or urls.py.

Valid options are:

HAYSTACK_SITECONF = 'myproject.search_sites'

No default is provided.

3.10.3 HAYSTACK_SEARCH_ENGINE

Required
This setting controls which backend should be used. You should provide the short name (e.g. solr), not the full filename of the backend (e.g. solr_backend.py).

Valid options are:

HAYSTACK_SEARCH_ENGINE = 'solr'
HAYSTACK_SEARCH_ENGINE = 'whoosh'
HAYSTACK_SEARCH_ENGINE = 'dummy'

No default is provided.

3.10.4 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.
3.10.5 HAYSTACK_INCLUDE_SPELLING

Optional
This setting controls if spelling suggestions should be included in search results. This can potentially have performance implications so it is disabled by default.

An example:

HAYSTACK_INCLUDE_SPELLING = True

Works for the solr and whoosh backends.

3.10.6 HAYSTACK_SOLR_URL

Required when using the ‘solr’ backend
This setting controls what URL the solr backend should be connecting to. This depends on how the user sets up their Solr daemon.

Examples:

HAYSTACK_SOLR_URL = 'http://localhost:9000/solr/test'
HAYSTACK_SOLR_URL = 'http://solr.mydomain.com/solr/mysite'

No default is provided.

3.10.7 HAYSTACK_SOLR_TIMEOUT

Optional when using the ‘solr’ backend
This setting controls the time to wait for a response from Solr in seconds.

Examples:

HAYSTACK_SOLR_TIMEOUT = 30

The default is 10 seconds.

3.10.8 HAYSTACK_WHOOSH_PATH

Required when using the ‘whoosh’ backend
This setting controls where on the filesystem the Whoosh indexes will be stored. The user must have the appropriate permissions for reading and writing to this directory.

Any trailing slashes should be left off.

Finally, you should ensure that this directory is not located within the document root of your site and that you take appropriate security precautions.

An example:

HAYSTACK_WHOOSH_PATH = '/home/mysite/whoosh_index'

No default is provided.
3.10.9 HAYSTACK_XAPIAN_PATH

Required when using the ‘xapian’ backend

This setting controls where on the filesystem the Xapian indexes will be stored. The user must have the appropriate permissions for reading and writing to this directory.

Any trailing slashes should be left off.

Finally, you should ensure that this directory is not located within the document root of your site and that you take appropriate security precautions.

An example:

HAYSTACK_XAPIAN_PATH = '/home/mysite/xapian_index'

No default is provided.

3.10.10 HAYSTACK_BATCH_SIZE

Optional

This setting controls the number of model instances loaded at a time while reindexing. This affects how often the search indexes must merge (an intensive operation).

An example:

HAYSTACK_BATCH_SIZE = 100

The default is 1000 models per commit.

3.10.11 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.

3.11 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>.
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 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'
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.

### 4.1.2 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'
django-admin.py test solr_tests --settings=solr_settings
```

Or, to run the Whoosh backend’s tests:

```
cd django-haystack/tests
export PYTHONPATH='pwd'
django-admin.py test whoosh_tests --settings=whoosh_settings
```
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.
CHAPTER 5

Requirements

Haystack has a relatively easily-met set of requirements.

- Python 2.4+ (may work on 2.3 but untested)
- Django 1.0+

Additionally, each backend has its own requirements. You should refer to *Installing Search Engines* for more details.