RestPose Python bindings

Release 0.7.0

Richard Boulton

August 08, 2011

CONTENTS

RestPose is an easy-to-use search server, built on top of Xapian. This module is a Python client for it, allowing access to all the features provided by RestPose in a natural way.

CHAPTER

ONE

OVERVIEW

So, let's suppose you have some documents that you want to search. The first thing to do is to get the RestPose server installed and running; see the RestPose documentation on *Installation* for details of this.

Then, you can install the RestPose python client (we suggest in a virtualenv), using:

```
$ pip install restpose
```

Indexing some documents is as simple as:

```
import restpose
server = restpose.Server('http://localhost:7777')
doc = { 'text': 'Hello world', 'tag': 'A tag' }
coll = server.collection("test_coll")
coll.add_doc(doc, doc_type="blurb", doc_id=1)
```

And then, searching those documents can be done by:

```
>>> s = coll.doc_type("blurb").field("text").text("Hello")
>>> s.matches_estimated
1
>>> s[0].data == {'text': ['Hello world'],
...               'tag': ['A tag'],
...               'type': ['blurb'],
...               'id': ['1']}
True
```

The rest of this overview goes through some of the subtle details and extra features we just skipped over in that example.

1.1 Connecting to the server

Once the server is running, it provides a REST API over HTTP. The server runs on port 7777 by default, and for the rest of this tutorial we'll assume that it's running on the same machine as you're using the python client on, and is running on the default port.

```
from restpose import Server
server = Server('http://localhost:7777')
```

1.2 Adding a document

As far as RestPose is concerned, a document is a set of fields, each of which has a name, and each of which has one or more values. Documents also have types and IDs. The document type is used to determine how each field should be interpreted; the configuration of how to index and search each field can be specified for each document type (though you will often be able to use RestPose's default configuration). The ID is used to identify the document, and is unique within a given type.

IDs must be specified when sending a document to the RestPose server; the python client currently doesn't automatically allocate IDs if they are missing.

Documents are stored within collections, which are just named groups of documents. It is not currently possible to search transparently across multiple collections. Collections should be used when you have independent projects, but wish to share the resources of a server across them.

```
doc = { 'text': 'Hello world', 'tag': 'A tag' }
coll = server.collection("test_coll")
coll.add_doc(doc, doc_type="blurb", doc_id=1)
```

If all goes well, within a short time (usually a fraction of a second), the document will have been indexed. However, using the above calls the changes won't be fully applied until a few seconds later (by default, until 5 seconds of inactivity), and the new document will be available for searching until this has occurred. This delay is deliberate, and is to allow bulk updates to be performed efficiently, but can be avoided using a checkpoint.

1.3 Checkpoints

Documents are added asynchronously; it's important to realise that the `add_doc` function will only report an error if it is unable to insert the document into the indexing queue on the server (eg, because the server is down, or overloaded). It will not report an error if the document is invalid in some way, and the document will not immediately be available for searching.

In addition, documents are processed in parallel; if I add document A and then add document B, it is quite possible for processing of document B to finish before processing of document A.

There is of course, a way to check for errors, to ensure the ordering of particular modification operations and also to ensure that changes are made ready for searching without the usual wait for inactivity. These tasks are all performed using `CheckPoints`.

```
>>> checkpt = coll.checkpoint().wait()
>>> checkpt.total_errors, checkpt.errors, checkpt.reached
(0, [], True)
```

Essentially, what's happening here is that the checkpoint is put into the indexing queue in such a way that it will be processed only when all tasks placed onto the queue before it have been completed, and that it will be processed before any tasks placed onto the query after it are started. When it is processed, the preceding changes are committed (ie, made available for searching).

The `wait` method blocks until the checkpoint has been processed. Alternatively, if you don't want to block, the checkpt.reached property will reflect the current state of the checkpoint on the server.

Note: Currently, the server doesn't support long-polling for checkpoint status, so the wait() method is implemented by polling the server periodically. This implementation is likely to be improved in future.

It is also possible to make a checkpoint which doesn't cause a commit, in order to collect errors and control ordering of processing operations. To do this, simply pass *commit=False* to the `Collection.checkpoint` method when creating the checkpoint.

1.4 Field types

Todo

This section needs rewriting for clarity (sorry!).

There are many different ways in which the data supplied in a field can be processed and made available for searching. The way in which each field is indexed is controlled by the collection configuration, and can be adjusted separately for each document type.

Essentially, the configuration maps each field name to a field type, and associates various parameters with those field types.

Additionally, when a new field is seen (ie, one for which the configuration doesn't have an entry), the configuration contains a list of patterns which are applied in order, and the first match is used to configure the new field.

Currently, the RestPose Python client doesn't provide much support to help you work with collection configuration; it just provides a mechanism for getting and setting the full configuration as a hierarchical structure. The full configuration of a collection may be obtained from the server using the the `Collection.config` property. This may then be modifed and applied back to the property. For example, to add a pattern to the default configuration for a new document type (ie, to the configuration which will be used when a new document type is seen for the first time):

```
>>> c = coll.config
>>> c['default_type']['patterns'].insert(0,
...     ['test',
...      {'group': 'T', 'max_length': 10, 'too_long_action': 'error',
...       'type': 'exact'}])
>>> coll.config = c
```

After the above, adding a new document to the collection with a previously unseen document type would cause the configuration for indexing the document type to process a field called "test" for exact matching, in group "T", but produce an error if any entries in the "test" field were longer than 10 characters.

Details of the field types available, the parameters which can be applied to them, and the default list of patterns, are contained in the server documentation: *Types and Schemas*.

1.5 Searching

There are several ways to build up and perform a search in RestPose. Here's a simple example:

```
>>> search = coll.field('text').text('Hello')
>>> print len(search)
1
>>> search[0].data
{'text': ['Hello world'], 'tag': ['A tag'], 'type': ['blurb'], 'id': ['1']}
```

By convention, the word `query` is used in RestPose to refer to a set of operations which can be used to match and associate a weight with a set of documents. The word `search` is then used as a noun to refer to an object comprising a query, and any other options involved in performing the search (for example, the offset of the first result to retrieve

from the server, or options controlling additional information to retrieve). The word `search` is also used as a very to refer to the operation of performing a search.

Queries can be constructed in several ways. Firstly, a query can be created which searches the contents of a named field.

```
>>> query = coll.field.text.parse('Hello')
```

In this case, `query` will represent a query on the "text" field, and will use the query parser configured for that field to build a query from the word "Hello". The query will also be bound to the collection `coll`, so that when it is performed, the entire collection will be searched. We say that the target of the query is the entire collection.

Note: if the field name is not a valid python identifier, or is stored in a variable, you can use an alternative syntax of calling the `coll.field` property, passing the field name as a parameter. For example:

```
>>> query = coll.field('text').parse('Hello')
```

A query can also be created which is bound to a document type within a collection; when such a query is performed, only documents of the given type will be considered for matching. For example, the following command will produce a query which has a target of the "blurb" document type within the collection.

```
>>> query = coll.doc_type('blurb').field('text').text('Hello')
```

A query can also be created which is bound to neither a document type nor a collection; before such a query can be performed it must be given a target (which can be done by combining it with a query which is already associated with a target, or by explicitly setting a target using the `set_target` method.).

```
>>> from restpose import Field
>>> query = Field('text').text('Hello')
```

What's happening behind the scenes here is that the `field` method and the `Field` factory produce a `FieldQuerySource` object, which provides various methods for creating queries.

Some query types can be performed across all fields; for this, the `AnyField` factory can be used to create unbound queries, or the `any_field` method can be used to create bound queries on collections or document types. The documentation for each query type indicates whether it is valid to search across all fields with that query type.

1.5.1 Primitive query types

There are several "primitive" query types other than the "text" type described so far. Most of these are only applicable to fields which have been configured in particular ways. For full details of the search options available in RestPose, see the server documentation on *Searches*; this section will discuss how to construct each type of query in Python.

- `FieldQuerySource.`**`is_in`**`(`*values*`)`
  Create a query for fields which exactly match the given values.

  A document will match if at least one of the stored values for the field exactly matches at least one of the given values.

  This query type is currently available only for "exact", "id" and "cat" field types.

  Parameters **value** – A container holding the values to search for. As a special case, if a string is supplied, this is equivalent to supplying a container holding that string.

  Example Search for documents in which the "tag" field has a value of "edam", "cheddar" or "leicester".

```
>>> query = coll.field.tag.is_in(['edam', 'cheddar', 'leicester'])
```

Search for documents in which the "tag" field has a value of "edam".

```
>>> query = coll.field.tag.is_in('edam')
```

- `FieldQuerySource.`**`equals`**(*value*)
  Create a query for fields which exactly match the given value.

  Matches documents in which the supplied value exactly matches the stored value.

  This query type is currently available only for "exact", "id" and "cat" field types.

  This query type may be constructed using the == operator, or the `equals` method.

  Parameters **value** – The value to search for.

  Example Search for documents in which the "tag" field has a value of "edam".

  ```
  >>> query = coll.field.tag.equals('edam')
  ```

  Or, equivalently (but less conveniently for chained calls)

  ```
  >>> query = (coll.field.tag == 'edam')
  ```

- `FieldQuerySource.`**`range`**(*begin*, *end*)
  Create a query for field values in a given range.

  Matches documents in which one of the stored values in the field are in the specified range, including both the begin and end values.

  This type is currently available only for "double", "date" and "timestamp" field types.

  Parameters

  - **begin** – The start of the range.
  - **end** – The end of the range.

  Example Search for documents in which the "num" field has a value in the range 0 to 10 (including the endpoints).

  ```
  >>> query = coll.field.num.range(0, 10)
  ```

- `FieldQuerySource.`**`text`**(*text*, *op='phrase'*, *window=None*)
  Create a query for a piece of text in the field.

  This is a simple search for a matching sequences of words (subject to whatever processing has been performed on the field to conflate variant forms of words, such as stemming or word splitting for CJK text).

  Parameters

  - **text** – The text to search for. If empty, this query will match no results.
  - **op** – The operator to use when searching. One of "or", "and", "phrase" (ordered proximity), "near" (unordered proximity). Default="phrase".
  - **window** – Only relevant if op is "phrase" or "near". Window size in words within which the words in the text need to occur for a document to match; None=length of text. Integer or None. Default=None

Example Search for documents in which the "text" field contains text matching the phrase "Hello world".

```
>>> query = coll.field.text.text("Hello world")
```

- `FieldQuerySource.`**`parse`**(*text*, *op='and'*)
  Parse a structured query, searching the field.

  Unlike text, this allows various operators to be used in the query; for example, parentheses may be used, and operators such as "AND" may be used

  ---

  Todo

  Document the operators permitted.

  Beware that the parser is unable to make sense of some query strings (eg, those with mismatched parentheses). If such a query string is used, an error will be returned by the server when the search is performed.

  Parameters

  - **fieldname** – The field to search within.
  - **text** – Text to search for. If empty, this query will match no results.
  - **op** – The default operator to use when searching. One of "or", "and". Default="and".

  Example Search for documents in which the "text" field contains both "Hello" and "world", but not "big".

  ```
  >>> query = coll.field.text.text("Hello world -big")
  ```

- `FieldQuerySource.`**`exists`**()
  Search for documents in which the field exists.

  This type may be used to search across all fields.

  Example Search for documents in which the "text" field exists.

  ```
  >>> query = coll.field.text.exists()
  ```

  Search for documents in which any field exists.

  ```
  >>> query = coll.any_field.exists()
  ```

- `FieldQuerySource.`**`nonempty`**()
  Search for documents in which the field has a non-empty value.

  This type may be used to search across all fields.

  Example Search for documents in which the "text" field has a non-empty value.

  ```
  >>> query = coll.field.text.nonempty()
  ```

  Search for documents in which any field has a non-empty value.

  ```
  >>> query = coll.any_field.nonempty()
  ```

- `FieldQuerySource.`**`empty`**`()`
  Search for documents in which the field has an empty value.

  This type may be used to search across all fields.

  Example Search for documents in which the "text" field has an empty value.

  ```
  >>> query = coll.field.text.empty()
  ```

  Search for documents in which any field has an empty value.

  ```
  >>> query = coll.any_field.empty()
  ```

- `FieldQuerySource.`**`has_error`**`()`
  Search for documents in which the field produced errors when parsing.

  This type may be used to search across all fields.

  Example Search for documents in which the "text" field had an error when parsing.

  ```
  >>> query = coll.field.text.has_error()
  ```

  Search for documents in which any field had an error when parsing.

  ```
  >>> query = coll.any_field.has_error()
  ```

Note: it is perfectly possible to construct a query on a field which cannot be performed due to the way in which a field has been configured; many queries can only be performed on certain types of field. If you do this, you'll get an error when you try to perform the search, not when you construct the query.

There are also a couple of primitive query types which aren't specific to a field, and can be created by methods of `Collection` or `DocumentType`.

- `QueryTarget.`**`all`**`()`
  Create a query which matches all documents.
- `QueryTarget.`**`none`**`()`
  Create a query which matches no documents.

1.5.2 Combining queries

Queries can be combined using several operators to build a query tree. These operators can be used to produce various boolean combinations of matching results, and also to influence the way in which weights are combined.

There are many ways in which queries can be combined; the simplest to describe are the boolean operations:

- Boolean AND

  A query can be constructed which only returns documents which match all of a set of subqueries.

  class `restpose.query.`**`And`**`(`**queries*, ***kwargs*`)`
  A query which matches only the documents matched by all subqueries.

  The weights are the sum of the weights in the subqueries.

  Example A query returning documents in which the `tag` field contains both the value `'foo'` and the value `'bar'`.

```
>>> query = And(Field('tag').equals('foo'),
...             Field('tag').equals('bar'))
```

Such a query can also be constructed by joining two queries with the `&` operator:

`Query.__and__`(*other*)
Produce an And query combining this query with `other`.

Parameters **other** – The query to combine with this query.

Example A query returning documents in which the `tag` field contains both the value `'foo'` and the value `'bar'`.

```
>>> query = Field('tag').equals('foo') & Field('tag').equals('bar')
```

- Boolean OR

A query can be constructed which only returns documents which match all of a set of subqueries.

class `restpose.query.Or`(**queries*, ***kwargs*)
A query which matches the documents matched by any subquery.

The weights are the sum of the weights in the subqueries which match.

Example A query returning documents in which the `tag` field contains at least one of the value `'foo'` or the value `'bar'`.

```
>>> query = Or(Field('tag').equals('foo'),
...            Field('tag').equals('bar'))
```

Such a query can also be constructed by joining two queries with the `|` operator:

`Query.__or__`(*other*)
Produce an Or query combining this query with `other`.

Parameters **other** – The query to combine with this query.

Example A query returning documents in which the `tag` field contains at least one of the value `'foo'` or the value `'bar'`.

```
>>> query = Field('tag').equals('foo') | Field('tag').equals('bar')
```

- Boolean AND-NOT

Rather than supporting a unary NOT operator (which would return all documents not matched by a query), RestPose implement an "AndNot" operator, which returns documents which match one query, but do not match another query.

The lack of a unary NOT operator is because it is not generally possible to efficiently compute a list of all the documents in a Collection which do not match a query with the datastructures in use by RestPose. Also, because it is difficult to associate useful scores with documents matching a unary NOT operator, it is rarely desirable to implement a unary NOT operator. If you really need a unary NOT, you can use an `all` query as part of the AndNot operator.

To construct a query which returns documents which match one query, but do not match any of a set of other queries:

class `restpose.query.AndNot`(**queries*, ***kwargs*)
A query which matches the documents matched by the first subquery, but not any of the other subqueries.

The weights returned are the weights in the first subquery.

Example A query returning documents in which the `tag` field contains the value `'foo'` but not the value `'bar'`.

```
>>> query = AndNot(Field('tag').equals('foo'),
...                Field('tag').equals('bar'))
```

Such a query can also be constructed by joining two queries with the − operator:

`Query.__sub__`(*other*)
Produce an AndNot query combining this query with `other`.

Parameters **other** – The query to combine with this query.

Example A query returning documents in which the `tag` field contains the value `'foo'` and not the value `'bar'`.

```
>>> query = Field('tag').equals('foo') - Field('tag').equals('bar')
```

- Filter

A filter query is a query which returns documents and weights from an initial query, but removes any documents which do not match another query (or set of queries).

The `Filter` constructor allows a query to be constructed which returns documents which match all of a set of subqueries, but only returns the weight from the first of these subqueries.

class `restpose.query.Filter`(**queries*, ***kwargs*)
A query which matches the documents matched by all the subqueries, but only returns weights from the first subquery.

Example A query returning documents in which the `tag` field contains the value `'foo'`, with weights from this match, but only where the `tag` field also contains the value `'bar'`.

```
>>> query = Filter(Field('tag').equals('foo'),
...                Field('tag').equals('bar'))
```

Such a query can also be constructed by joining two queries with the `filter` method:

`Query.filter`(*other*)
Return the results of this query filtered by another query.

This returns only documents which match both the original and the filter query, but uses only the weights from the original query.

Parameters **other** – The query to combine with this query.

Example A query returning documents in which the `tag` field contains the value `'foo'`, filtered to only include documents in which the `tag` field also contains the value `'bar'`.

```
>>> query = Field('tag').equals('foo').filter(Field('tag').equals('bar'))
```

- AndMaybe

An AndMaybe query is a query which returns only those documents which match an initial query, but adds weights from a set of other subqueries. This can be used to adjust weights based on external factors (for example, matching tags), without causing extra documents to match the query.

The `AndMaybe` constructor allows a query to be constructed which returns documents which match all of a set of subqueries, but only returns the weight from the first of these subqueries.

class restpose.query.**AndMaybe**(**queries*, ***kwargs*)
A query which matches the documents matched by the first subquery, but adds additional weights from the other subqueries.

The weights are the sum of the weights in the subqueries.

Example A query returning documents in which the tag field contains the value 'foo', with weights from this match, but with additional weights for any of these documents in which the tag field contains the value 'bar'.

```
>>> query = AndMaybe(Field('tag').equals('foo'),
...                  Field('tag').equals('bar'))
```

Such a query can also be constructed by joining two queries with the and_maybe method:

Query.**and_maybe**(*other*)
Return the results of this query, with additional weights from another query.

This returns exactly the documents which match the original query, but adds the weight from corresponding matches to the other query.

Parameters **other** – The query to combine with this query.

Example A query returning documents in which the tag field contains the value 'foo', but with additional weights for any matches containing the value 'bar'.

```
>>> query = Field('tag').equals('foo').and_maybe(Field('tag').equals('bar'))
```

- Weight multiplication and division

The weights returned from a query can be modified by multiplying them by a constant (positive) factor. This can be used to bias the results from part of a combined query over the results from other parts of a combined query.

The MultWeight constructor allows a query to be constructed which returns exactly the same documents as a subquery, but with the weight multiplied by a factor.

class restpose.query.**MultWeight**(*query*, *factor*, *target=None*)
A query which matches all the documents matched by another query, but with the weights multiplied by a factor.

Example A query returning documents in which the tag field contains the value 'foo', with weights multiplied by 2.5.

```
>>> query = MultWeight(Field('tag').equals('foo'), 2.5)
```

Build a query in which the weights are multiplied by a factor.

Such a query can also be constructed by use of the * operator, applied to a positive number and a query (the query may be either on the right hand or left hand side):

Query.**__mul__**(*mult*)
Return a query with the weights scaled by a multiplier.

This can be used to build a query in which the weights of some subqueries are increased or decreased relative to the other subqueries.

Parameters **mult** – The multiplier to apply. Must be a positive number.

Example A query returning documents in which the tag field contains the value 'foo' and the weights are multiplied by 2.5

```
>>> query = Field('tag').equals('foo') * 2.5
```

Weights can also be divided using the / operator.

- Boolean XOR

  Finally, RestPose also supports an XOR operator - this is rarely of much practical use, but is included for completeness of boolean operators.

  A query can be constructed which only returns documents which match an odd number of a set of subqueries.

  class restpose.query.**Xor**(**queries*, ***kwargs*)
  A query which matches the documents matched by an odd number of subqueries.

  The weights are the sum of the weights in the subqueries which match.

  Example A query returning documents in which the tag field contains exactly one of the value 'foo' or the value 'bar'.

  ```
  >>> query = Xor(Field('tag').equals('foo'),
  ...             Field('tag').equals('bar'))
  ```

  Such a query can also be constructed by joining two queries with the ^ operator:

  Query.**__xor__**(*other*)
  Produce an Xor query combining this query with other.

  Parameters **other** – The query to combine with this query.

  Example A query returning documents in which the tag field contains exactly one of the value 'foo' or the value 'bar'.

  ```
  >>> query = Field('tag').equals('foo') ^ Field('tag').equals('bar')
  ```

1.5.3 Performing searches

Now you've done all this work to get a query, you'll almost certainly want to perform a search using it. Fortunately, this is very easy.

If you wish to control exactly when a search is sent to the server, you can perform a search directly using the search method on a Collection or DocumentType. This returns a SearchResults object which provides convenient access to the results as returned from the server.

However, an alternative approach which is often more convenient is also provided: Query objects can be sliced and subscripted to get at the list of matching documents. They also support various methods and properties to get statistics about things like the number of matching documents. Communication with the server will be performed when necessary, and the results of such communication will be cached.

For example, suppose we have a query such as:

```
>>> query = coll.field('text').text('Hello')
```

To get the first result of a query:

```
>>> print query[0]
SearchResult(rank=0, data={'text': ['Hello world'], 'tag': ['A tag'], 'type': ['blurb'], 'id': ['1']}
```

Suppose you want the top 10 results. One approach would be to subscript the query with 0, 1, 2, etc. This will actually be fairly efficient - the Python RestPose client will request pages of results when it doesn't know how many results you're going to want (the default page size is 20, but this can be adjusted by changing the `page_size` property). You can even iterate over all matching documents using the standard python iteration mechanism; the iterator will return SearchResult objects.

```
>>> for r in query: print r
SearchResult(rank=0, data={'text': ['Hello world'], 'tag': ['A tag'], 'type': ['blurb'], 'id': ['1']}
```

To get just the first 10 results of the query, you can slice the query; this returns a `TerminalQuery`, which has all the same properties for performing searches as the other Query classes we've discussed so far, but may not be combined with other Query objects. The TerminalQuery can be subscripted and iterated, but (unless the slice has an open upper end) you are guaranteed that the results will be requested from the server in a single page of size and offset governed by the slice. `Query` and `TerminalQuery` have a common base class of `Searchable`.

The Xapian search engine, used by RestPose, implements some sophisticated optimisations for calculating the top results of a query without having to calculate all the possible documents matching a query. To give these optimisations as much scope to work as possible, you should usually slice your query before accessing individual search results.

To get the total number of matching documents, you can use the `len` builtin on a Query object. This will cause a search to be performed if necessary, and will return the exact number of matching documents. However, again, if you do not need the exact number of matching documents, you can allow the Xapian optimisations to work a lot better by using a set of properties which produce an estimate and bounds on the number of matching documents. Specifically:

- the `matches_lower_bound` property returns a lower bound on the number of matching documents.
- the `matches_estimated` property returns an estimate of the number of matching documents.
- the `matches_upper_bound` property returns an upper bound on the number of matching documents.
- the `estimate_is_exact` property returns True if the estimate produced by `matches_estimated` is known to be the exact number of matching documents.

It is possible to influence how much work Xapian performs when searching to calculate the number of matching documents. This can be done using the `check_at_least` method, which produces a new `TerminalQuery` with the supplied `check_at_least` value. When the search is performed, Xapian will check at least this number of documents for being potential matches to the search (if there are sufficient matches). This ensures that the estimate and bounds will be exact if fewer documents match than the supplied number; higher `check_at_least` values will increase the accuracy of the estimate, but will reduce the speed at which the search is performed.

Setting the `check_at_least` value can also be useful when calculating additional match information, such as counting term occurrence, and faceting.

Another useful property is the `total_docs` property, which returns the number of documents in the target of the search (ie, in the DocumentType or Collection searched).

1.6 Additional information (Facets, Term occurrence)

Often, it is useful to be able to get additional information along with a search result; for example, in a faceted search application, it is desirable to get counts of the number of matching documents which have each value of a field, which are then used to display options for narrowing down the search.

Currently, RestPose supports getting two types of additional information: occurrence counts for terms, and co-occurrence counts for terms. While the occurrence counts feature could be used to support a faceted search interface, it wouldn't perform particularly well, because it is fairly slow to access the term occurrence counts. More efficient support for faceted search (involving storing the required information in a slot allowing for faster access) will be implemented in a future release; if you have urgent need of it, contact the author on IRC (in #restpose on irc.freenode.net).

1.6.1 Term occurrence

RestPose can calculate counts of each term seen in matching documents. To do this, use *calc_occur* on the Searchable to indicate that the information should be calculated during the search.

`Searchable.`**`calc_occur`**(*prefix*, *doc_limit=None*, *result_limit=None*, *get_termfreqs=False*, *stopwords=[]*)

Get occurrence counts of terms in the matching documents.

Warning - fairly slow.

Causes the search results to contain counts for each term seen, in decreasing order of occurrence. The count entries are of the form: [suffix, occurrence count] or [suffix, occurrence count, termfreq] if get_termfreqs was true.

Parameters

- **prefix** – prefix of terms to check occurrence for
- **doc_limit** – number of matching documents to stop checking after. None=unlimited. Integer or None. Default=None
- **result_limit** – number of terms to return results for. None=unlimited. Integer or None. Default=None
- **get_termfreqs** – set to true to also get frequencies of terms in the db. Boolean. Default=False
- **stopwords** – list of stopwords - term suffixes to ignore. Array of strings. Default=[]

The occurrence counts can then be retrieved via the `Searchable.info` property. Note that it is usually advisable to set the `check_at_least` value for such a search, to ensure that a reasonable number of potential matches will be included when calculating the occurrence counts. Conversely, because calculating this requires access to the termlists for each document observed, which is a slow operation, the *calc_occur* method allows you to limit the number of documents checked using the *doc_limit* parameter; you can set this to get a sampling of the documents in the index, rather than potentially checking all of them (note that such a sampling isn't unbiased, unfortunately; the documents which are sampled will be the ones nearer the start of the index, which usually means those documents which were indexed first).

1.6.2 Term co-occurrence

Similarly, Restpose can calculate counts of which term-pairs occur together most often. To do this, use *calc_cooccur* on the Searchable to indicate that the information should be calculated during the search.

`Searchable.`**`calc_cooccur`**(*prefix*, *doc_limit=None*, *result_limit=None*, *get_termfreqs=False*, *stopwords=[]*)

Get cooccurrence counts of terms in the matching documents.

Warning - fairly slow (and O(L*L), where L is the average document length).

Causes the search results to contain counts for each pair of terms seen, in decreasing order of cooccurrence. The count entries are of the form: [suffix1, suffix2, co-occurrence count] or [suffix1, suffix2, co-occurrence count, termfreq of suffix1, termfreq of suffix2] if get_termfreqs was true.

Parameters

- **prefix** – prefix of terms to check co-occurrence for
- **doc_limit** – number of matching documents to stop checking after. None=unlimited. Integer or None. Default=None

- **result_limit** – number of terms to return results for. None=unlimited. Integer or None. Default=None
- **get_termfreqs** – set to true to also get frequencies of terms in the db. Boolean. Default=False
- **stopwords** – list of stopwords - term suffixes to ignore. Array of strings. Default=[]

The same options as when calculating term occurrence counts apply for controlling the number of documents considered when calculating this information. Note that calculating this is significantly more expensive than calculating the pure occurrence counts, so in a large system you might well want to start with small limits, and gradually increase the counts until performance is no longer acceptable.

CHAPTER

TWO

RESTPOSE MODULES

2.1 Client

The RestPose client mirrors the resources provided by the RestPose server as Python objects.

class `restpose.client.`**`Server`**(*uri='http://127.0.0.1:7777'*, *resource_class=None*, *resource_instance=None*, ***client_opts*)

Representation of a RestPose server.

Allows indexing, searching, status management, etc.

Parameters

- **uri** – Full URI to the top path of the server.
- **resource_class** – If specified, defines a resource class to use instead of the default class. This should usually be a subclass of `RestPoseResource`.
- **resource_instance** – If specified, defines a resource instance to use instead of making one with the default class (or the class specified by *resource_class*.
- **client_opts** – Parameters to use to update the existing client_opts in the resource (if *resource_instance* is specified), or to use when creating the resource (if *resource_class* is specified).

`status`

Get server status.

Returns a dictionary holding the status as returned from the server. See the server documentation for details.

`collections`

Get a list of existing collections.

Returns a list of collection names (as strings).

`collection`(*coll_name*)

Access to a collection.

Parameters **coll_name** – The name of the collection to access.

Returns a Collection object which can be used to search and modify the contents of the Collection.

Note: No request is performed directly by this method; a Collection object is simply created which will make requests when needed. For this reason, no error will be reported at this stage even if the collection does not exist, or if a collection name containing invalid characters is used.

class restpose.client.**FieldQueryFactory**(*target=None*)
Object for creating searches on a field.

Parameters **target** – The target to pass to the Query objects created.

target
The target that will be used when creating Query objects. Defaults to None.

class restpose.client.**FieldQuerySource**(*fieldname*, *target=None*)
An object which generates queries for a specific field.

Parameters

- **fieldname** – The name of the field to generate queries for. If set to None, will generate queries across all fields.
- **target** – The target to generate queries pointing to.

is_in(*values*)
Create a query for fields which exactly match the given values.

A document will match if at least one of the stored values for the field exactly matches at least one of the given values.

This query type is currently available only for "exact", "id" and "cat" field types.

Parameters **value** – A container holding the values to search for. As a special case, if a string is supplied, this is equivalent to supplying a container holding that string.

Example Search for documents in which the "tag" field has a value of "edam", "cheddar" or "leicester".

```
>>> query = coll.field.tag.is_in(['edam', 'cheddar', 'leicester'])
```

Search for documents in which the "tag" field has a value of "edam".

```
>>> query = coll.field.tag.is_in('edam')
```

equals(*value*)
Create a query for fields which exactly match the given value.

Matches documents in which the supplied value exactly matches the stored value.

This query type is currently available only for "exact", "id" and "cat" field types.

This query type may be constructed using the == operator, or the `equals` method.

Parameters **value** – The value to search for.

Example Search for documents in which the "tag" field has a value of "edam".

```
>>> query = coll.field.tag.equals('edam')
```

Or, equivalently (but less conveniently for chained calls)

```
>>> query = (coll.field.tag == 'edam')
```

range(*begin*, *end*)
Create a query for field values in a given range.

Matches documents in which one of the stored values in the field are in the specified range, including both the begin and end values.

This type is currently available only for "double", "date" and "timestamp" field types.

Parameters

- **begin** – The start of the range.
- **end** – The end of the range.

Example Search for documents in which the "num" field has a value in the range 0 to 10 (including the endpoints).

```
>>> query = coll.field.num.range(0, 10)
```

text (*text*, *op='phrase'*, *window=None*)

Create a query for a piece of text in the field.

This is a simple search for a matching sequences of words (subject to whatever processing has been performed on the field to conflate variant forms of words, such as stemming or word splitting for CJK text).

Parameters

- **text** – The text to search for. If empty, this query will match no results.
- **op** – The operator to use when searching. One of "or", "and", "phrase" (ordered proximity), "near" (unordered proximity). Default="phrase".
- **window** – Only relevant if op is "phrase" or "near". Window size in words within which the words in the text need to occur for a document to match; None=length of text. Integer or None. Default=None

Example Search for documents in which the "text" field contains text matching the phrase "Hello world".

```
>>> query = coll.field.text.text("Hello world")
```

parse (*text*, *op='and'*)

Parse a structured query, searching the field.

Unlike text, this allows various operators to be used in the query; for example, parentheses may be used, and operators such as "AND" may be used

---

Todo

Document the operators permitted.

Beware that the parser is unable to make sense of some query strings (eg, those with mismatched parentheses). If such a query string is used, an error will be returned by the server when the search is performed.

Parameters

- **fieldname** – The field to search within.
- **text** – Text to search for. If empty, this query will match no results.
- **op** – The default operator to use when searching. One of "or", "and". Default="and".

Example Search for documents in which the "text" field contains both "Hello" and "world", but not "big".

```
>>> query = coll.field.text.text("Hello world -big")
```

exists()
Search for documents in which the field exists.

This type may be used to search across all fields.

Example Search for documents in which the "text" field exists.

```
>>> query = coll.field.text.exists()
```

Search for documents in which any field exists.

```
>>> query = coll.any_field.exists()
```

nonempty()
Search for documents in which the field has a non-empty value.

This type may be used to search across all fields.

Example Search for documents in which the "text" field has a non-empty value.

```
>>> query = coll.field.text.nonempty()
```

Search for documents in which any field has a non-empty value.

```
>>> query = coll.any_field.nonempty()
```

empty()
Search for documents in which the field has an empty value.

This type may be used to search across all fields.

Example Search for documents in which the "text" field has an empty value.

```
>>> query = coll.field.text.empty()
```

Search for documents in which any field has an empty value.

```
>>> query = coll.any_field.empty()
```

has_error()
Search for documents in which the field produced errors when parsing.

This type may be used to search across all fields.

Example Search for documents in which the "text" field had an error when parsing.

```
>>> query = coll.field.text.has_error()
```

Search for documents in which any field had an error when parsing.

```
>>> query = coll.any_field.has_error()
```

class restpose.client.**QueryTarget**
An object which can be used to make and run queries.

field
Factory for field-specific queries.

any_field
Pseudo field for making queries across all fields.

all()
Create a query which matches all documents.

none()
Create a query which matches no documents.

find(*q*)
Apply a Query to this QueryTarget.

Parameters **q** – A Query object which will have the target applied to it.

search(*search*)
Perform a search.

Parameters **search** – is a search structure to be sent to the server, or a Search or Query object.

class restpose.client.**Document**(*collection*, *doc_type*, *doc_id*)

data

terms

values

class restpose.client.**DocumentType**(*collection*, *doc_type*)

add_doc(*doc*, *doc_id=None*)
Add a document to the collection.

delete_doc(*doc_id*)
Delete a document with this type from the collection.

get_doc(*doc_id*)

class restpose.client.**Collection**(*server*, *coll_name*)

doc_type(*doc_type*)

status
The status of the collection.

config
The configuration of the collection.

add_doc(*doc*, *doc_type=None*, *doc_id=None*)
Add a document to the collection.

delete_doc(*doc_type*, *doc_id*)
Delete a document from the collection.

get_doc(*doc_type*, *doc_id*)
Get a document from the collection.

checkpoint(*commit=True*)
Set a checkpoint on the collection.

This creates a resource on the server which can be queried to detect whether indexing has reached the checkpoint yet. All updates sent before the checkpoint will be processed before indexing reaches the checkpoint, and no updates sent after the checkpoint will be processed before indexing reaches the checkpoint.

delete()
Delete the entire collection.

class restpose.client.**CheckPoint**(*collection*, *check_id*)
A checkpoint, used to check the progress of indexing.

check_id
The ID of the checkpoint.

This is used to identify the checkpoint on the server.

reached
Return true if the checkpoint has been reached.

May contact the server to check the current state.

Raises CheckPointExpiredError if the checkpoint expired before the state was checked.

errors
Return the list of errors associated with the CheckPoint.

Note that if there are many errors, only the first few will be returned.

Returns None if the checkpoint hasn't been reached yet.

Raises CheckPointExpiredError if the checkpoint expired before the state was checked.

total_errors
Return the total count of errors associated with the CheckPoint.

This may be larger than len(self.errors), if there were more errors than the CheckPoint is able to hold.

Returns None if the checkpoint hasn't been reached yet.

Raises CheckPointExpiredError if the checkpoint expired before the state was checked.

wait()
Wait for the checkpoint to be reached.

This will contact the server, and wait until the checkpoint has been reached.

If the checkpoint expires (before or during the call), a CheckPointExpiredError will be raised. Otherwise, this will return the checkpoint, so that further methods can be chained on it.

2.2 Query

Queries in RestPose.

class restpose.query.**Searchable**(*target*)
An object which can be sliced or iterated to perform a query.

Create a new Searchable.

target is the object that the search will be performed on. For example, a restpose.Collection or restpose.DocumentType object.

page_size
Number of results to get in each request, if size is not explicitly set.

set_target(*target*)
Return a searchable, with the target set.

If the target was already set to the same value, returns self. Otherwise, returns a copy of target.

search()
Explicitly force a search for this query to be performed.

This ignores any cached results, and always makes a call to the server.

The query should usually be sliced before calling this method. If the slice does not specify an endpoint, the server will use its internal limit on the number of results, so only a small number of results will be returned unless a larger number is explictly set by slicing.

Returns The results of the search.

total_docs
Get the total number of documents searched.

matches_lower_bound
A lower bound on the number of matches.

matches_estimated
An estimate of the number of matches.

matches_upper_bound
An upper bound on the number of matches.

estimate_is_exact
True if the value returned by matches_estimated is exact, False if it isn't (or at least, isn't guaranteed to be).

check_at_least(*check_at_least*)
Set the check_at_least value.

This is the minimum number of documents to try and check when running the search - useful mainly when you want reasonably accurate counts of matching documents, but don't want to retrieve all matches.

Returns a new Search, with the check_at_least value to use when performing the search set to the specified value.

order_by(*field*, *ascending=None*)
Set the sort order.

info
Get the list of information items returned by the search.

calc_occur(*prefix*, *doc_limit=None*, *result_limit=None*, *get_termfreqs=False*, *stopwords=*[])
Get occurrence counts of terms in the matching documents.

Warning - fairly slow.

Causes the search results to contain counts for each term seen, in decreasing order of occurrence. The count entries are of the form: [suffix, occurrence count] or [suffix, occurrence count, termfreq] if get_termfreqs was true.

Parameters

- **prefix** – prefix of terms to check occurrence for
- **doc_limit** – number of matching documents to stop checking after. None=unlimited. Integer or None. Default=None
- **result_limit** – number of terms to return results for. None=unlimited. Integer or None. Default=None
- **get_termfreqs** – set to true to also get frequencies of terms in the db. Boolean. Default=False
- **stopwords** – list of stopwords - term suffixes to ignore. Array of strings. Default=[]

calc_cooccur(*prefix*, *doc_limit=None*, *result_limit=None*, *get_termfreqs=False*, *stopwords=[]*)
Get cooccurrence counts of terms in the matching documents.

Warning - fairly slow (and O(L*L), where L is the average document length).

Causes the search results to contain counts for each pair of terms seen, in decreasing order of cooccurrence. The count entries are of the form: [suffix1, suffix2, co-occurrence count] or [suffix1, suffix2, co-occurrence count, termfreq of suffix1, termfreq of suffix2] if get_termfreqs was true.

Parameters

- **prefix** – prefix of terms to check co-occurrence for
- **doc_limit** – number of matching documents to stop checking after. None=unlimited. Integer or None. Default=None
- **result_limit** – number of terms to return results for. None=unlimited. Integer or None. Default=None
- **get_termfreqs** – set to true to also get frequencies of terms in the db. Boolean. Default=False
- **stopwords** – list of stopwords - term suffixes to ignore. Array of strings. Default=[]

class restpose.query.**QueryIterator**(*query*)
Iterate over the results of a query.

next()

class restpose.query.**Query**(*target=None*)
Base class of all queries.

All query subclasses should have a property called "_query", containing the query as a structure ready to be converted to JSON and sent to the server.

filter(*other*)
Return the results of this query filtered by another query.

This returns only documents which match both the original and the filter query, but uses only the weights from the original query.

Parameters **other** – The query to combine with this query.

Example A query returning documents in which the `tag` field contains the value `'foo'`, filtered to only include documents in which the `tag` field also contains the value `'bar'`.

```
>>> query = Field('tag').equals('foo').filter(Field('tag').equals('bar'))
```

and_maybe(*other*)
Return the results of this query, with additional weights from another query.

This returns exactly the documents which match the original query, but adds the weight from corresponding matches to the other query.

Parameters **other** – The query to combine with this query.

Example A query returning documents in which the `tag` field contains the value `'foo'`, but with additional weights for any matches containing the value `'bar'`.

```
>>> query = Field('tag').equals('foo').and_maybe(Field('tag').equals('bar'))
```

class restpose.query.**QueryField**(*fieldname*, *querytype*, *value*, *target=None*)
A query in a particular field.

class restpose.query.**QueryMeta**(*querytype*, *value*, *target=None*)
A query for meta information (about field presence, errors, etc).

class restpose.query.**QueryAll**(*target=None*)
A query which matches all documents.

class restpose.query.**QueryNone**(*target=None*)
A query which matches no documents.

restpose.query.**QueryNothing**
alias of QueryNone

class restpose.query.**CombinedQuery**(**queries*, ***kwargs*)
Base class of Queries which are combinations of a sequence of queries.

Subclasses must define self._op, the operator to use to combine queries.

class restpose.query.**And**(**queries*, ***kwargs*)
A query which matches only the documents matched by all subqueries.

The weights are the sum of the weights in the subqueries.

Example A query returning documents in which the `tag` field contains both the value `'foo'` and the value `'bar'`.

```
>>> query = And(Field('tag').equals('foo'),
...             Field('tag').equals('bar'))
```

class restpose.query.**Or**(**queries*, ***kwargs*)
A query which matches the documents matched by any subquery.

The weights are the sum of the weights in the subqueries which match.

Example A query returning documents in which the `tag` field contains at least one of the value `'foo'` or the value `'bar'`.

```
>>> query = Or(Field('tag').equals('foo'),
...            Field('tag').equals('bar'))
```

class restpose.query.**Xor**(**queries*, ***kwargs*)
A query which matches the documents matched by an odd number of subqueries.

The weights are the sum of the weights in the subqueries which match.

Example A query returning documents in which the `tag` field contains exactly one of the value `'foo'` or the value `'bar'`.

```
>>> query = Xor(Field('tag').equals('foo'),
...             Field('tag').equals('bar'))
```

class restpose.query.**AndNot**(**queries*, ***kwargs*)
A query which matches the documents matched by the first subquery, but not any of the other subqueries.

The weights returned are the weights in the first subquery.

Example A query returning documents in which the `tag` field contains the value `'foo'` but not the value `'bar'`.

```
>>> query = AndNot(Field('tag').equals('foo'),
...                Field('tag').equals('bar'))
```

class restpose.query.**Filter**(**queries*, ***kwargs*)
A query which matches the documents matched by all the subqueries, but only returns weights from the first subquery.

Example A query returning documents in which the `tag` field contains the value `'foo'`, with weights from this match, but only where the `tag` field also contains the value `'bar'`.

```
>>> query = Filter(Field('tag').equals('foo'),
...                Field('tag').equals('bar'))
```

class restpose.query.**AndMaybe**(**queries*, ***kwargs*)
A query which matches the documents matched by the first subquery, but adds additional weights from the other subqueries.

The weights are the sum of the weights in the subqueries.

Example A query returning documents in which the `tag` field contains the value `'foo'`, with weights from this match, but with additional weights for any of these documents in which the `tag` field contains the value `'bar'`.

```
>>> query = AndMaybe(Field('tag').equals('foo'),
...                  Field('tag').equals('bar'))
```

class restpose.query.**MultWeight**(*query*, *factor*, *target=None*)
A query which matches all the documents matched by another query, but with the weights multiplied by a factor.

Example A query returning documents in which the `tag` field contains the value `'foo'`, with weights multiplied by 2.5.

```
>>> query = MultWeight(Field('tag').equals('foo'), 2.5)
```

Build a query in which the weights are multiplied by a factor.

class restpose.query.**TerminalQuery**(*orig*, *slice=None*)
A Query which has had offsets or additional search options set.

This is produced from a Query when additional search options are set. It can't be combined with other Query objects, since the semantics of doing so would be confusing.

class restpose.query.**SearchResult**(*rank*, *data*)

class restpose.query.**SearchResults**(*raw*)
The results returned from the server when performing a search.

total_docs
The total number of documents searched.

offset
The offset of the first result item.

size_requested
The requested size.

check_at_least
The requested check_at_least value.

matches_lower_bound
A lower bound on the number of matches.

matches_estimated
An estimate of the number of matches.

matches_upper_bound
An upper bound on the number of matches.

estimate_is_exact
Return True if the value returned by matches_estimated is exact, False if it isn't (or at least, isn't guaranteed to be).

items
The matching result items.

info
The list of information items returned from the server.

at_rank(*rank*)
Get the result at a given rank.

The rank is the position in the entire result set, starting at 0.

Raises IndexError if the rank is out of the range in the result set.

2.3 Errors

Errors specific to RestPose.

exception restpose.errors.**RestPoseError**

exception restpose.errors.**CheckPointExpiredError**
An error raised when a checkpoint has expired.

2.4 Resource

Resources for RestPose.

This module provides a convenient interface to the resources exposed via HTTP by the RestPose server.

class restpose.resource.**RestPoseResponse**(*connection*, *request*, *resp*)
A response from the RestPose server.

In addition to the properties exposed by restkit:restkit.Response, this exposes a *json* property, to decode JSON responses automatically.

json
Get the response body as JSON.

Returns The response body as a python object, decoded from JSON, if the response Content-Type was application/json.

Raises an exception if the Content-Type is not application/json, or the body is not valid JSON.

Raises RestPoseError if the status code returned is not one of the supplied status codes.

expect_status(**expected*)
Check that the status code is one of a set of expected codes.

Parameters **expected** – The expected status codes.

Raises RestPoseError if the status code returned is not one of the supplied status codes.

class restpose.resource.**RestPoseResource**(*uri*, ***client_opts*)
A resource providing access to a RestPose server.

This may be subclassed and provided to restpose.Server, to allow requests to be monitored or modified. For example, a logging subclass could be used to record requests and their responses.

Initialise the resource.

Parameters

- **uri** – The full URI for the resource.
- **client_opts** – Any options to be passed to restkit.Resource.

user_agent
The user agent to send when making requests.

request(*method*, *path=None*, *payload=None*, *headers=None*, ***params*)
Perform a request.

Parameters

- **method** – the HTTP method to use, as a string.
- **path** – The path to request.
- **payload** – A payload to send as the request body; may be a file-like object, or a string, or a structure to send encoded as a JSON object.
- **headers** – A dictionary of headers. If not already set, Accept and User-Agent headers will be added to this, and if there is a JSON payload, the Content-Type will be set to application/json.
- **params_dict** – A dictionary of parameters to add to the requst URI.

CHAPTER

THREE

DOCUMENTATION TODOS

Todo

Document the operators permitted.

(The *original entry* is located in modules.rst, line 224.)

Todo

This section needs rewriting for clarity (sorry!).

(The *original entry* is located in overview.rst, line 147.)

Todo

Document the operators permitted.

(The *original entry* is located in overview.rst, line 12.)

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

r