RestPose Documentation

Release 0.7.6

Richard Boulton

November 10, 2011

CONTENTS

Contents:

CHAPTER

ONE

OVERVIEW

RestPose is a search engine. By this, we mean that it is a system which is designed to take a set of documents, and then when given a query to return ranked lists of documents which are a good match for that query. RestPose manages a set of internal indexes, and provides an interface (over HTTP, in a fairly RESTful style, using JSON as the main transfer format) which allows documents to be submitted and removed from indexes, and which allows searches to be performed.

RestPose includes support for processing text in a variety of languages (the most spoken European languages, and CJK text, have explicit support, but other text can often be handled to some level). It also has some basic, but reasonably effective, support for automatically guessing the language of a piece of text.

RestPose is developed under Linux and MacOSX, but should also be possible to compile for windows with a small amount of work.

1.1 What is the point of a search engine?

With most database systems (certainly with relational databases, and also with many of the newer "NoSQL" databases), the focus of queries performed on a database is to retrieve a set of records matching the query. In contrast, the main focus of queries performed in a search engine is to retrive a ranked list of records; and often, only to return the most highly ranked. Whilst a good ranking is important, it is hard to define exactly what a good ranking is, so there are many ways to customise how the ranking is performed. It is generally a binary yes/no decision to tell if a database is returning the correct answer to a query; with a search engine, it is much less clear cut what the correct answer to a query is, so the appropriate topic to discuss is often how high quality the results of a query are - ie, how well the ranking matches some theoretcial ideal.

Many databases provide some support for "full text search" these days. This is often sufficient to implement a very simple search engine, but doesn't allow the flexibility to tune ranking in the ways that a search engine allows.

Similarly, it is possible to use a search engine as your primary datastore, at the cost of losing some control over aspects such as fully transactional updates, ability to perform complex joins, etc.

The difference of focus between a database and a search engine often means that a good approach is to use both a standard database for some types of query, and a search engine for others.

1.2 Development

RestPose is still in early development - it is likely that the internal storage formats will change from time to time (though this will be noted clearly in release notes), and the APIs are likely to be modified as

Development is performed using Git, using github to hold the master repository. The Python client is actively maintained by the same author as the core engine, since it is used to run parts of the main testsuite.

Issues can be reported in the git repository at http://github.com/rboulton/restpose

CHAPTER

TWO

INSTALLATION

Please report any problems in this document with particular architectures. So far, RestPose has been built on Linux (Ubuntu 11.04 and Fedora 8) and Mac OSX ("Snow Leopard") without problems.

2.1 Binary packages

Sorry, there are no binary packages available yet.

2.2 Compilation from source

The following are instructions for building from a source distribution tarball.

2.2.1 Installing prerequisites on Fedora 8

Build and install a recent Xapian first:

```
sudo yum install -y gcc-c++ zlib-devel e2fsprogs-devel
wget http://oligarchy.co.uk/xapian/1.2.5/xapian-core-1.2.5.tar.gz
tar zxf xapian-core-1.2.5.tar.gz
cd xapian-core-1.2.5
./configure
make
sudo make install
sudo vi /etc/ld.so.conf # Add /usr/local/lib to end of file
sudo ldconfig
```

2.2.2 Installing prerequisites on Ubuntu

Install the Xapian library and development headers, a compiler, and some necessary dependency libraries, using:

```
sudo apt-get install build-essential libxapian-dev python zlib1g-dev uuid-dev
```

2.2.3 Compiling RestPose

Then, build RestPose, simply by downloading and unpacking a distribution tarball, and running:

```
./configure
make
```

This produces an executable "restpose", which is used for both indexing and searching.

Note that, if instead of a distribution tarball, you are working from a checkout from a version control system, you will need to have GNU autotools installed, and run:

```
./bootstrap
```

to build the build system and configure script.

2.2.4 Building documentation

In order to build the documentation, you will need the Python "Sphinx" package installed. On Ubuntu, for example, the package can be installed using:

```
sudo apt-get install python-sphinx
```

However, you often won't need to build your own copies of the documentation; distribution tarballs contain a copy, and copies of the documentation are available in various places; in particular, a copy of the latest revision is available at http://readthedocs.org/docs/restpose/en/latest/

2.2.5 Running tests

Once you have built RestPose, you can build and run the testsuite simply by running:

```
make check
```

(In the source directory.) Note that this will also build and run the testsuite for libmicrohttpd, which may fail on some machines due to ports used by the testsuite being in use, or due to firewalling rules. Instead, you might like to run:

```
make check-am
```

which will only run the tests for RestPose.

CHAPTER

THREE

COLLECTIONS

Everything in RestPose is stored in a *Collection*. There may be many collections in a RestPose instance, each with entirely distinct contents.

The main objects stored in a collection are *Documents*. These each have a *type*, which governs how they are indexed, stored and searched, and an *ID*, which is used to reference a specific document for updates and retrieval.

Collections also contain:

A *Schema* for each type of document. This controls how each field in the document is handled when indexing and searching. In addition, the schema contains a set of patterns used to determine how to handle new fields - whether to raise an error for them, or add an appropriate field configuration for them.

A set of *Categorisers* These can can be applied to parts of a document for tasks such as identifying the language of a piece of text.

A set of *Taxonomies*

Collections contain a named set of taxonomies, each of which contains a set of categories. The schema can associate fields with taxonomies. Searches can then efficiently match all items in a given category, or in the children of a given category.

A set of *Orderings* Documents in the collection, or a subset of documents in the collection, may be placed in a specific order. Search results may then be sorted by this ordering, and the position of a particular document in the ordering may be determined efficiently.

A set of *Pipes* These allow custom manipulation of documents to be performed, with the output sent to a subsequent pipe, or to the index. Note - this subsystem doesn't feel elegant and clean, and may therefore be entirely replaced at some point in the near future by an alternative approach, such as Lua scripting.

Collections are identified by name, which is assumed to be a UTF-8 encoded value when referred to in URIs, and is not allowed to contain the following characters:

- "Control" characters: ie, characters in the range 0 to 31.
- `:`, `/`, `\`, `.`, `,`, `[`, `]`, `{`, `}`

3.1 Documents

As far as RestPose is concerned, a document is a object consisting of an unordered set of fieldnames, each of which has a value, or an ordered list of values, associated with it. In JSON representation, a document will often look something like:

```
{
  "id": "1",
  "type": "default",
  "text": ["Lorem ipsum", "Dolor sit"],
  "tag": ["Simple", "Boring tag"]
}
```

Documents will usually have two fields which are used for special purposes by RestPose: an ID field, and a Type field. The ID field is used for finding documents with a specific ID, and the Type field is used for finding documents with a specific type.

IDs are specific to a particular type of document - ie, two documents may exist in the collection with the same ID if they have differing types. In other words, to uniquely reference a document in a collection, you need to use a combination of the type and the document ID.

Document IDs and type names are assumed to be UTF-8 encoded values when referred to in URIs, and are not allowed to contain the following characters:

- "Control" characters: ie, characters in the range 0 to 31.
- `:`, `/`, `\`, `.`, `,`, `[`, `]`, `{`, `}`

3.2 Types and Schemas

Each collection can contain a variety of types of document. The indexing and search configuration for each type is independent (with the exception that the same fields must be used for storing ids and type ids in every type), allowing entirely distinct indexing strategies to be used for each type.

Each type is described by a schema, which consists of several properties and governs the way in which search and indexing is performed:

fields A description of the configuration for each known field_type.

patterns A list of patterns to apply, in order, to unknown fields.

The collection configuration contains a section which defines which fields are used to hold identifiers for document ids and document types.

3.2.1 Field types

The list of known fields is stored in the "*fields*" property of a schema. This maps from a fieldname to a set of properties for that field. The configuration for all fields must have a "*type*" property, which defines what kind of processing to do to the field. All field types except the "Ignore" type also accept a "*store_field*" parameter, which specifies the fieldname to store values seen in that field in (for display with search results). This will normally be either empty if the values shouldn't be stored, or the fieldname for the field.

The other properties which are valid vary according to the type. There are several different types:

Text fields

(*type* = *text*)

Text fields expect a UTF-8 encoded string as input.

They have a *group* parameter used to distinguish the terms generated by these fields from other fields. The *group* parameter may not be empty.

They also have a *processor* parameter used to control how terms are generated from words. Currently supported values for this parameter are:

- (empty): Standard Xapian term processing, with no stemming. Essentially, this means: lowercase, split on whitespace and punctuation.
- "stem_*": Standard Xapian term processing, using the stemming algorithm matched by "*". Eg, "stem_en" to use the English stemming algorithm.
- "cjk": Processing for CJK text, splitting text into ngrams. Non CJK characters are split on whitespace.

Exact fields

(*type* = *exact*)

Exact fields expect a single byte string, or a single integer, as input, and store a representation of the exact value supplied as input.

Exact fields have a *group* parameter which is used to distinguish the terms generated by these fields from other fields. The *group* parameter may not be empty.

If an integer is supplied (either when indexing or searching), it is converted to a decimal representation of the integer (as long as the integer is positive, and requires no more than 64 bits to represent it in binary form).

Exact fields have a *wdfinc* parameter, which specifies the frequency to store for each occurrence of a term. This must be an integer, and defaults to 0, which means that terms will be stored with a frequency in documents of 0, no matter how often they occur in a document. This is an appropriate value for fields which are only ever used for filtering, but if you wish to get a higher weight for searches which contain multiple matching terms in them, you should set it to at least 1. Higher values will cause documents matching terms in this field to get a higher weight.

Exact fields have a *max_length* parameter, which specifies the maximum length for a field value to be stored. This defaults to 64. With current Xapian backends there is a limit on term length - to avoid any possible problems, this limit shouldn't be raised above 120 (though you can get away with larger values in many cases). It's probably unwise to have very long terms anyway.

Exact fields also have a *too_long_action* parameter, which specifies an action to take if the field value exceeds the length specified by *max_length*. This can be one of:

- *error*: Log an error if the field value exceeds the maximum length. This is the default.
- *hash*: Replace the end of the field value with a hash of the end of the field value, to bring the length into compliance with *max_length*. Note that if *max_length* is very low (a few bytes), the hashed length might still exceed it.
- *truncate*: Truncate the field value to the *max_length* value.

Exact fiels also have a *lowercase* parameter. This defaults to `false`, but if it is set to `true` it causes the text supplied to the field to be lowercased before being used to generate terms. It has no effect on the contents stored in the document data, or in the slot for the field. It will also be used at search time to lowercase terms generated in the query, so searches will be case insensitive.

Numeric (double) fields

(*type* = *double*)

Numeric fields expect a numeric value, which will be stored as a double precision floating point value. Precision loss may occur if the numeric values supplied cannot be represented as a double precision floating point value (but note that, for example, all 32 bit integer values can be accurately represented as doubles).

They have one additional parameter: the "slot" parameter, which is the number or name of the slot that the values will be stored in. Each numeric field that should be searchable should be given a distinct value for the "slot" parameter. See the slot_numbers section for more details about slot numbers.

Category fields

(*type* = *cat*)

Category fields are somewhat similar to exact fields, but are attached to a taxonomy (essentially, a hierarchy of field values). Searches can then be used to find all documents in which a value in a document is a descendant of the search value.

They have one additional parameter: the "taxonomy" parameter, which is the name of the taxonomy used by the field. Multiple independent fields may make use of the same taxononmy.

In order to work correctly, it is advisable to ensure that the term *group* used for a category field is not shared with any other fields which use a different taxonomy, or which are not category fields.

Each field value may be given one or more parents. It is also possible for a parent to have multiple child values. It is an error to attempt to set up loops in the inheritance graph, however.

See the Taxonomies section for more details.

Timestamp fields

(*type* = *timestamp*)

Timestamp fields expect an integer number of seconds since the Unix epoch (1970). They can only handle positive values.

They have one additional parameter: the "slot" parameter, which is the number or name of the slot that the timestamps will be stored in. Each timestamp that should be searchable should be given a distinct value for the "slot" parameter. See the slot_numbers section for more details about slot numbers.

Date fields

(*type* = *date*)

Date fields expect a date in the form "year-month-day", in which year, month and day are integer values. Negative years are allowed.

They have one additional parameter: the "slot" parameter, which is the number or name of the slot that dates will be stored in. Each date that should be searchable should be given a distinct value for the "slot" parameter. See the slot_numbers section for more details about slot numbers.

LonLat fields (geospatial)

(*type* = *lonlat*)

LonLat fields expect to be given geographic coordinates (as longitude, latitude pairs, expressed in degrees). Multiple coordinates may be given in a field; each coordinate may be expressed either as an array of numbers of length 2 (in which case, the longitude must be given first), or as an object with "lon" and "lat" properties, holding the longitude and latitude respectively, as numbers.

They have one additional parameter: the "slot" parameter, which is the number or name of the slot that the coordinates will be stored in. Each field containing coordinates that should be separately searchable should be given a distinct value for the "slot" parameter. See the slot_numbers section for more details about slot numbers.

LonLat fields allow searches to be performed which:

- return only documents within a given range.
- return results in order of closest distance from a set of points.
- return results in a combined order of distance and other scores from the query.

Stored fields

(*type = stored*)

Stored fields do nothing except store their input value for display. They have no additional parameters.

Ignore fields

(*type = ignore*)

Ignore fields are completely ignored. They may be defined to prevent the default action for unknown fields being performed on them. Note that this field type does not support the *store_field* parameter; the contents of an ignored field will never be stored.

ID fields

(*type = id*)

ID fields expect a single byte string as input.

There should only be one ID field in a schema. This field is used to generate a unique ID for the documents; if a new document is added with the same ID, the old document with that ID will be replaced.

ID fields are very similar to Exact fields - they accept all the same parameters, with the exception of the *group* parameter.

Meta fields

(*type = meta*)

Meta fields are a special field type. A normal field shouldn't be assigned a meta field type. The meta field is used to store information about which fields were present in a document, and which fields produced errors when processing. It can then be used to search for these values.

Slot numbers

Various fields (eg, timestamp and date fields) have a "slot" parameter in their configuration. This is related to a concept in Xapian called "value slots" - each document can have values associated with it, to be used at search time for filtering, sorting, etc.

Xapian has a limitation that the slots are addressed only by numbers rather than by strings. For convenience, restpose allows slots to be addressed by strings, and hashes the strings to produce a number. There is a small chance of hash collision, but this is unlikely to be a problem unless you are using

It is still possible to use a number to reference a slot. If you use a number directly, it is advisable to use a number less than 0x0fffffff, since the results of the hashing algorithm will always be in the range 0x10000000 to 0xffffffff.

Note that the string *"1"* is not the same as the number 1. The string will be hashed to produce a slot number, whereas the number will be used directly as the slot number.

Groups

Many field types (eg, text and exact fields) have a "group" parameter in their configuration. These parameters hold a single string, which is the name of the group to generate terms in for that field.

When indexing, these fields generate terms representing the content of the fields. In order that distinct fields do not generate the same terms, the terms can be placed into separate groups. This allows searches to specify which field a piece of content must be found in. However, sometimes you want distinct fields to be merged when performing searches; in this case, you can specify the same group for several fields.

In most cases, you should take care that all fields which share a group use a compatible indexing strategy; ie, they should have the same field type, and have the same values for configuration parameters other than those controlling the frequencies stored (eg, the wdfinc). If you really know what you are doing, though, it is valid for entirely different configurations to share a group - you just might not get the results you expected.

Note: With the current search engine backend, short values should be used for the group parameters, since longer values will use quite a bit more space. A future backend database format is expected to make this difference minimal, but for now, simply use short names if you're concerned about disk usage (a few characters should be fine).

3.2.2 Patterns

The "*patterns*" property of a schema contains a list of patterns which are used to define new field types automatically the first time a new field is seen.

Each pattern is a list containing two items: the pattern to match, and the type definition to use when the pattern matches.

When indexing, for any field which is not already in the list of known fields in the schema the patterns are checked, in order, against the name of the new field. The first matching value is then used to create a new field type.

Because document processing happens in parallel, it is important that the order of processing documents is not significant in controlling what the new field's configuration should be. Therefore, only the name of the new field is taken into account; the contents of the field in the document being processed are not significant.

Currently, the only syntax supported for patterns is a literal fieldname with an optional leading "*". If present, the "*" will match any number of characters (including 0) at the start of the fieldname.

A "*" character may be used in the values of type definitions to use when a pattern matches. This will be replaced by whatever the "*" matched in the pattern.

3.2.3 Special fields

The "*special_fields*" property of a collection defines the field names which are used for special purposes.

id_field The field which is used for id lookups. This should normally be a field of type id. The terms generated from the id field will be used for replacing older versions of documents.

type_field The field which is used for type lookups.

meta_field The field which is used for storing meta information about which fields are present in documents, and which fields have errors. This should be a field of type meta. Incoming documents should not contain entries in the meta field - the entries will be automatically generated based on the result of processing the documents.

3.3 Taxonomies

A collection may contain a set of Taxonomies, each identified by a name. These taxonomies consist of a hierarchy of categories, and can be associated with fields. When associated with a category field, it becomes possible to search efficiently for all documents in which a field value is a descendant of a given category; normally, it would only be possible to do this by constructing a huge query consisting of all the categories which are such descendants.

Taxonomies may be modifed at any time by adding or removing category-parent relationships, or even hold categories. The necessary index updates will be performed automatically.

Note: If possible, it is better to put the hierarchy in place before adding documents, since this will require less work in total.

Note: Currently, the taxonomy feature is not designed to perform well with large numbers of entries in the category hierarchy (ie, more than a few hundred entries). Performance improvements are planned, but if you need to use the feature with deep hierarchies, contact the author on IRC or email.

3.4 Categorisers

3.5 Orderings

3.6 Pipes

CHAPTER

FOUR

COLLECTION CONFIGURATION

An example dump of the configuration for a collection can be found in examples/schema.json.

The schema is a JSON file (with the extension that C-style comments are permitted in it). The current schema format is stored in a *schema_format* property: this is to allow upgrades to the schema format to be performed in future. This document describes schemas for which *schema_format* is 3.

Todo

Describe the representation of the collection configuration fully.

CHAPTER

FIVE

CATEGORISATION

RestPose currently offers a very straightforward categorisation system, aimed mainly at performing language guessing (though it would be easy to plug more advanced algorithms into the system in future). The system requires a sample of text for each target category, and generates an ordered list of ngrams which are most significant in that sample.

The algorithm used is the same as that used by "libtextcat", and based on a 1994 paper entitled "N-gram-based text categorization" by William B. Cavnar and John M. Trenkle. It is a very simple algorithm, but processes text extremely fast and performs the language guessing text extremely well (given suitable training data). At the time of writing, the paper can be downloaded from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.21.3248&rep=rep1&type=pdf

CHAPTER

SIX

TRAINING

To train a model, you need some sample data. One approach for language guessing is to download some sample data from wikipedia, which can be done manually, or can be done using the script in *scripts/get_wikipedia_sample_text.py*. This script will download 512k of text from random wikipedia pages in each language supported by RestPose, and store it in a directory named *lang_samples*. This sample data can then be used to train the model.

Once you have sample data, the "restpose" program can be used on the command line to build a model, producing a JSON description of the model on stdout. This can be quite long, so you'll probably want to redirect the output to a file. For example:

```
./restpose -a train -d lang_samples -l en -l fr
```

The above command will train a model using the sample data for english and french. To add each language, add a "-l lang" parameter, where lang is the language code for the language to add.

For convenience, the command to train a model using all the data downloaded by the *scripts/get_wikipedia_sample_text.py* script is:

```
./restpose -a train -d lang_samples -l da -l de -l en -l es -l fi -l fr -l hu -l it -l ja -l ko -l nl
```

CHAPTER

SEVEN

RESTPOSE URL STRUCTURE

This section of the RestPose documentation details the URLs at which RestPose makes resources available.

7.1 General notes

7.1.1 Parameter types

There are three ways to pass parameters in HTTP requests:

- As part of the request path (referred to in the following documentation as *Parameters*).
- As part of the request query string (referred to in the following documentation as *Query Parameters*).
- As part of the request body. Where this is applicable, the details of the body to supply is described below.

7.1.2 Boolean parameters

When a parameter is documented as accepting a boolean value, "true" can be represented as *1*, *true*, *yes* or *on*, and "false" can be represented as *0*, *false*, *no* or *off* (with all strings comparisons here being case insensitive).

7.1.3 C-style escapes

Where non-unicode strings need to be returned, they will be escaped using C-style escaping. More precisely:

- \ will be represented as \\
- Tab (hex code 0x09) will be represented as \t
- Line feed (hex code 0x0a) will be represented as \n
- Carriage return (hex code 0x0d) will be represented as \r
- Characters from 0x32 to 0x7f (inclusive) will be represented as themselves.
- Any other characters will be represented as \xXX, where XX is the hex representation of the character code.

7.1.4 Error returns

If a 400 or 500 series HTTP error code is returned from the following methods, except where noted otherwise, the response body will be of type `application/json`, and will contain a JSON object with, at least, an `err` property, containing a string describing the error.

7.2 Collections

Everything to do with collections is available under the */coll* URL.

7.2.1 Listing collections

GET /coll
Get details of the collections held by the server.

Takes no parameters.

Status Codes

- **200** – Normal response: returns a JSON object keyed by collection name, in which the values are empty objects. (In future, some information about the collections may be available in the values.)

7.2.2 Information about an individual collection

GET /coll/(*collection_name*)

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.

On success, the return value is a JSON object with the following members:

- `doc_count`: The number of documents in the collection.

Status Codes

- **200** – If the collection exists, and no errors occur.
- **404** – If the collection does not exist. Returns a standard error object.

7.2.3 Deleting a collection

DELETE /coll/(*collection_name*)

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.

The collection of the given name is deleted, if it exists. If it doesn't exist, returns successfully, but doesn't change anything.

Status Codes

- **200** – Normal response: returns an empty JSON object.

7.2.4 Collection configuration

The collection configuration is represented as a JSON object; for details of its contents, see *Collection Configuration*.

GET /coll/(*collection_name*)**/config**
Get the collection configuration.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.

Status Codes

- **200** – Normal response: returns a JSON object representing the full configuration for the collection. See *Collection Configuration* for details.
- **404** – If the collection does not exist. Returns a standard error object.

PUT /coll/ (*collection_name*) **/config**
Set the collection configuration. Actually, adds a task to set the collection configuration to the processing queue. This may be monitored, waited for, and committed using checkpoints in just the same way as for the document addition APIs.

Creates the collection if it didn't exist before the call.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.

Status Codes

- **202** – Normal response: returns a JSON object representing the full configuration for the collection. See *Collection Configuration* for details.

7.2.5 Checkpoints

Checkpoints are used to control committing of changes, sequence order of modification operations, and also to allow a client to wait until changes have been applied.

Note that checkpoints will be removed automatically after a timeout (though by default this timeout is around 1 day, so this will rarely be an issue in practice).

Checkpoints also do not persist across server restarts.

GET /coll/ (*collection_name*) **/checkpoint**
Get details of the checkpoints which exist for a collection.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.

Status Codes

- **200** – Normal response: returns a JSON array of strings, each string is the ID of a checkpoint on the collection. If the collection doesn't exist, returns an empty array.

POST /coll/ (*collection_name*) **/checkpoint**
Create a checkpoint.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.

Query Parameters

- **commit** – (boolean). True if the checkpoint should cause a commit, False if the checkpoint should not cause a commit.

Status Codes

- **201** – Normal response: returns a JSON object containing a single item, with a key of `checkid` and a value being a string used to identify the newly created checkpoint. The `Location` HTTP header will be set to a URL at which the status of the checkpoint can be accessed.

GET /coll/(*collection_name*)**/checkpoint/**
checkpoint_id Get the status of a checkpoint. If the checkpoint doesn't exist (or has expired), or the collection doesn't exist, returns a null JSON value.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.
- **checkpoint_id** – The id of the checkpoint.

Status Codes

- **200** – If the checkpoint or collection doesn't exist, returns a null JSON value. Otherwise, returns a JSON object with three members:
  - *reached*: A boolean, true if the checkpoint has been reached, false otherwise. If false, no other members will exist in the JSON object.
  - *total_errors*: The number of errors which has occurred since the last error. Each error is a JSON object with the following members:
    - *msg*: A string holding the error message.
    - *doc_type*: The type of the document that was being processed when the error occurred, or an empty string if no document type is relevant.
    - *doc_id*: The ID of the document that was being processed when the error occurred, or an empty string if no document ID is relevant.
  - *errors*: An array of errors. If very many errors have occurred, only the top few will be returned.

7.2.6 Taxonomies and categories

A RestPose collection can contain a set of Taxonomies, each of which is identified by a name, and which contains a set of Categories. Each Category in a Taxonomy may be associated with other categories in parent-child relationships.

GET /coll/(*collection_name*)**/taxonomy**
Get a list of all taxonomies available in the collection.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.

Status Codes

- **200** – Returns a JSON array of strings, holding the names of the taxonomies in the collection.
- **404** – If the collection does not exist.

GET /coll/(*collection_name*)**/taxonomy/**
taxonomy_name Get details of the named taxonomy in the collection.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.
- **taxonomy_name** – The name of the taxonomy. May not contain `:/\.,` or tab characters.

Status Codes

- **200** – Returns a JSON object representing the contents of the taxonomy, mapping from category ID to an array of parent IDs.
- **404** – If the collection or taxonomy do not exist.

GET /coll/(*collection_name*)**/taxonomy/**
*taxonomy_name***/id/***cat_id* Get details of a category in a named taxonomy in the collection.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.
- **taxonomy_name** – The name of the taxonomy. May not contain `:/\.,` or tab characters.
- **cat_id** – The ID of the category. May not contain `:/\.,` or tab characters.

Status Codes

- **200** – Returns a JSON object representing the category in the taxonomy, indicating the relationships between that category and others.
- **404** – If the collection, taxonomy or category do not exist.

GET /coll/(*collection_name*)**/taxonomy/**
*taxonomy_name***/id/***cat_id***/parent/***parent_id* Check if a category has a given parent, in the named taxonomy in the collection.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.
- **taxonomy_name** – The name of the taxonomy. May not contain `:/\.,` or tab characters.
- **cat_id** – The ID of the category. May not contain `:/\.,` or tab characters.
- **parent_id** – The ID of the parent category. May not contain `:/\.,` or tab characters.

Status Codes

- **200** – Returns an empty JSON object if the parent supplied is a parent of the category supplied.
- **404** – If the collection, taxonomy, category or parent do not exist, or the parent is not a parent of the category.

GET /coll/(*collection_name*)**/taxonomy/**
*taxonomy_name***/top** Get the top level categories (ie, categories which have no parents) in the named taxonomy in the collection.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.
- **taxonomy_name** – The name of the taxonomy. May not contain `:/\.,` or tab characters.

Status Codes

- **200** – Returns a JSON object in which keys are the IDs of the top level categories, and the values are objects with two members:
  - child_count: (integer) number of child categories of this category.
  - descendant_count: (integer) number of descendant categories of this category.
- **404** – If the collection or taxonomy do not exist.

PUT /coll/(*collection_name*)**/taxonomy/**
*taxonomy_name***/id/***cat_id* Add a category, creating the collection, taxonomy and category if needed.

The body submitted is ignored.

Note: in future, it may be possible to set the parents of the category by supplying an appropriate structure in the body. For now, the parents need to be added with individual calls, so the body should be left empty.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.
- **taxonomy_name** – The name of the taxonomy. May not contain `:/\.,` or tab characters.
- **cat_id** – The ID of the category. May not contain `:/\.,` or tab characters.

Status Codes

- **202** – Normal response: returns a JSON object. This will usually be empty, but may contain the following:
  - `high_load`: contains an integer value of 1 if the processing queue is busy. Clients should reduce the rate at which they're sending documents is `high_load` messages persist.

PUT /coll/(*collection_name*)**/taxonomy/***taxonomy_name***/id/***cat_id***/parent/***parent_id* Add a parent to a category, creating the collection, taxonomy, category and parent if needed.

This will also update any documents which need to be updated to ensure that category searches still return the right answers.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.
- **taxonomy_name** – The name of the taxonomy. May not contain `:/\.,` or tab characters.
- **cat_id** – The ID of the category. May not contain `:/\.,` or tab characters.
- **parent_id** – The ID of the parent category. May not contain `:/\.,` or tab characters.

Status Codes

- **202** – Normal response: returns a JSON object. This will usually be empty, but may contain the following:
  - `high_load`: contains an integer value of 1 if the processing queue is busy. Clients should reduce the rate at which they're sending documents is `high_load` messages persist.

DELETE /coll/(*collection_name*)**/taxonomy/***taxonomy_name* Delete an entire taxonomy.

This will also update any documents which need to be updated as a result of there no longer being any category relationships in that taxonomy.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.
- **taxonomy_name** – The name of the taxonomy. May not contain `:/\.,` or tab characters.

Status Codes

- **202** – Normal response: returns a JSON object. This will usually be empty, but may contain the following:

- `high_load`: contains an integer value of 1 if the processing queue is busy. Clients should reduce the rate at which they're sending documents is `high_load` messages persist.

DELETE /coll/(*collection_name*)**/taxonomy/**
*taxonomy_name***/id/***cat_id* Remove a category. Will create the collection and taxonomy if they don't already exist.

This will also update any documents which need to be updated as a result of there no longer being any category relationships involving that category.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.
- **taxonomy_name** – The name of the taxonomy. May not contain `:/\.,` or tab characters.
- **cat_id** – The ID of the category. May not contain `:/\.,` or tab characters.

Status Codes

- **202** – Normal response: returns a JSON object. This will usually be empty, but may contain the following:
  - `high_load`: contains an integer value of 1 if the processing queue is busy. Clients should reduce the rate at which they're sending documents is `high_load` messages persist.

DELETE /coll/(*collection_name*)**/taxonomy/**
*taxonomy_name***/id/***cat_id***/parent/***parent_id* Remove a parent from a category. Will create the collection and taxonomy if they don't already exist.

This will also update any documents which need to be updated to ensure that category searches still return the right answers.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.
- **taxonomy_name** – The name of the taxonomy. May not contain `:/\.,` or tab characters.
- **cat_id** – The ID of the category. May not contain `:/\.,` or tab characters.
- **parent_id** – The ID of the parent category. May not contain `:/\.,` or tab characters.

Status Codes

- **202** – Normal response: returns a JSON object. This will usually be empty, but may contain the following:
  - `high_load`: contains an integer value of 1 if the processing queue is busy. Clients should reduce the rate at which they're sending documents is `high_load` messages persist.

7.2.7 Documents

GET /coll/(*collection_name*)**/type/**
*type***/id/***id* Get the stored information about the document of given ID and type.

Note that the information returned is not exactly the same as that supplied when the document was indexed: the returned information depends on the stored fields, but also includes the indexed information about the document.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.
- **type** – The type of the document.
- **id** – The ID of the document.

Status Codes

- **200** – Normal response: returns a JSON object representing the document. This object will have some or all of the following properties (properties for which the value would be empty are omitted).
  - data: A JSON object holding the stored fields, keyed by field name. Each value is an array of the values supplied for that field. Each item in the array of values may be any JSON value, depending on what was supplied when indexing the field.
  - terms: A JSON object holding the terms in the document. Each key is the string representation of a term (escaped using C-style escapes, since terms may be arbitrary binary values), in which the value is another JSON object with information about the occurrence of the term:
    - If the within-document-frequency of the term is non-zero, the *wdf* key will contain the within-document-frequency, as an integer.
    - If there are positions stored for the term, the *positions* key will contain an array of integer positions at which the term occurs.
  - values: A JSON object holding the values in the document. The keys in this object will be the slot numbers used, and the values will be a string holding a C-style escaped version of the data stored in the value slot.
- **404** – If the collection, type or document ID doesn't exist: returns a standard error object.

PUT /coll/(*collection_name*)**/type/***type***/id/***id*

Create, or update, a document with the given *collection_name*, *type* and *id*.

Creates the collection with default settings if it didn't exist before the call.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.
- **type** – The type of the document.
- **id** – The ID of the document.

Status Codes

- **202** – Normal response: returns a JSON object. This will usually be empty, but may contain the following:
  - `high_load`: contains an integer value of 1 if the processing queue is busy. Clients should reduce the rate at which they're sending documents is `high_load` messages persist.

POST /coll/(*collection_name*)**/type/***type*

Create, or update, a document with the given *collection_name* and *type*. The id of the document will be read from the document body, from the field configured in the collection configuration for storing IDs (by default, this is *id*).

Creates the collection with default settings if it didn't exist before the call.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.

- **type** – The type of the document.

Status Codes

- **202** – Normal response: returns a JSON object. This will usually be empty, but may contain the following:
  - `high_load`: contains an integer value of 1 if the processing queue is busy. Clients should reduce the rate at which they're sending documents is `high_load` messages persist.

POST /coll/(*collection_name*)**/id/**

id Create, or update, a document with the given *collection_name* and *id*. The type of the document will be read from the document body, from the field configured in the collection configuration for storing types (by default, this is *type*).

Creates the collection with default settings if it didn't exist before the call.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.
- **id** – The ID of the document.

Status Codes

- **202** – Normal response: returns a JSON object. This will usually be empty, but may contain the following:
  - `high_load`: contains an integer value of 1 if the processing queue is busy. Clients should reduce the rate at which they're sending documents is `high_load` messages persist.

POST /coll/(*collection_name*)

Create, or update, a document in the collection *collection_name*. The type and ID of the document will be read from the document body, from the fields configured in the collection configuration for storing types and IDs (by default, these are *type* and *id*).

Creates the collection with default settings if it didn't exist before the call.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.

Status Codes

- **202** – Normal response: returns a JSON object. This will usually be empty, but may contain the following:
  - `high_load`: contains an integer value of 1 if the processing queue is busy. Clients should reduce the rate at which they're sending documents is `high_load` messages persist.

DELETE /coll/(*collection_name*)**/type/***type***/id/***id*

Delete a document from a collection.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.
- **type** – The type of the document.
- **id** – The ID of the document.

Status Codes

- **202** – Normal response: returns a JSON object. This will usually be empty, but may contain the following:
  - `high_load`: contains an integer value of 1 if the processing queue is busy. Clients should reduce the rate at which they're sending documents is `high_load` messages persist.

7.2.8 Performing a search

Searches are performed by sending a JSON search structure in the request body. This may be done using a GET request, but will usually be done with a POST request, since not all software supports sending a body as part of a *GET* request.

`GET /coll/`(*collection_name*)**`/search`**

`POST /coll/`(*collection_name*)**`/search`**
Search for documents in a collection, across all document types.

The search is sent as a JSON structure in the request body: see the *Searches* section for details on the search structure.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.

Status Codes

- **200** – Returns the result of running the search, as a JSON structure. See the *Search results* section for details on the search result structure.
- **404** – If the collection is not found.

`GET /coll/`(*collection_name*)**`/type/`**
type`/search`

`POST /coll/`(*collection_name*)**`/type/`**
type`/search` Search for documents in a collection, and with a given document type.

The search is sent as a JSON structure in the request body: see the *Searches* section for details on the search structure.

Parameters

- **collection_name** – The name of the collection. May not contain `:/\.,` or tab characters.
- **type** – The type of the documents to search for.

Status Codes

- **200** – Returns the result of running the search, as a JSON structure. See the *Search results* section for details on the search result structure.
- **404** – If the collection is not found.

7.3 Getting the status of the server

`GET /status`
Gets details of the status of the server.

Status Codes

- **200** – Returns a JSON object, with the following items:
  - `tasks`: Details of the task processing queues and pools in progress. This is an object, with an entry for each named group of task queues in the system (eg, for "indexing", "processing" and "search"). Each entry is an object with the following members:
    * `queues`: Details of the status of the queues in the task queue group. This has an entry for each queue (for the "indexing" and "processing" groups, the name of each task queue is the name of the corresponding collection. For the "search" group, the name of each task queue corresponds to the name of the task being performed; eg, the task which produces this status output is in the "status" group, so there will always be a `status` entry in the `search` group). Each entry is an object with the following members:
      · `active`: (bool) True if the group is actively being processed. False if the group has been deactivated (this is usually done temporarily to avoid overloading target queues - eg, processing queues will be deactivated temporarily if their corresponding indexing queue exceeds a certain size. They will reactivate automatically when the indexing queue becomes less full).
      · `assigned`: (bool) True if a worker is assigned exclusively to this group. This is generally true for indexing tasks, but false for processing and search tasks.
      · `in_progress`: (int) The number of tasks in the group currently being processed.
      · `size`: (int) The number of tasks on the queue, waiting to be processed. This does not include the number of tasks actively being processed (ie, those counted by `in_progress`).
    * `threads`: Details of the threads in the thread pool for the queue. This has the following members:
      · `running`: (int) The number of running threads in the thread pool (including threads waiting for tasks).
      · `size`: (int) The number of threads owned by the pool (including threads shutting down).
      · `waiting_for_join`: (int) The number of threads in the pool waiting for cleanup after shutting down.

7.4 Root and static files

GET /

GET /static/(*static_path*)
Static files are served from the `static` directory. This is intended for hosting pretty web interfaces for server administration and management.

Status Codes

- **200** – the contents of the file. Note that the mimetype is guessed from the file extension, and only a very limited set of common extensions are known about currently.
- **404** – the file was not found.

CHAPTER

EIGHT

SEARCHES

A search is composed of a query, and various parameters to control which part of the set of matching documents to return, which fields to return in the matching documents, and which additional information about the matching documents to calculate and return.

```
SEARCH = {
    "query": QUERY,
    "from": <offset of first document to return.  Integer.  0 based.  Default=0>,
    "fromdoc": FROMDOC,
    "size": <maximum number of documents to return.  -1=return all matches.  Integer.  Default=10>,
    "check_at_least": <minimum number of documents to examine before early termination optimisations
    "info": [ INFO ],
    "order_by": ORDER_BY,
    "display": <list of fields to return>,
    "verbose": <flag indicating whether to return verbose debugging informat.  Boolean.  Default=fals
}
```

8.1 Basic queries

8.1.1 Matching everything, or nothing

These seem a little pointless on first sight, but are useful placeholders for slotting into a standard query structure, and occasionally have other uses. In many situations, these queries will be optimised away when performing the search.

A query which matches all documents:

```
QUERY = {
    "matchall": true
}
```

A query which matches no documents:

```
QUERY = {
    "matchnothing": true
}
```

8.1.2 A search for a value in a particular field

```
QUERY = {
    "field": [
        <fieldname>,
        <type of search to do>,
        <value to search for>
    ]
}
```

Various types of search are possible:

- "is": searches for documents in which a value stored in the field is equal to the value to search for. This type is available for "exact", "id" and "category" field types. The value to search for must be an array of values, each of which is either a string or an integer (in the range 0..2^64-1).
- "is_descendant": searches for documents in which a value stored in the field is a descendant of a value specified in the search. This type is available for "category" field types. The value to search for must be an array of values, each of which is either a string or an integer (in the range 0..2^64-1).
- "is_or_is_descendant": searches for documents in which a value stored in the field is a value specified in the search, or is a descendant of a value specified in the search. This type is available for "category" field types. The value to search for must be an array of values, each of which is either a string or an integer (in the range 0..2^64-1).

  This is produces an equivalent query to combining an "is" search with an "is_descendant" search using the "or" operator (though this version may be slightly faster to parse).
- "range": searches for documents in which a stored value is in a given range. This type is currently available only for "double", "date" and "timestamp" field types.
- "distscore": searches for documents within a given range of a center point, and returns a score which increases the closer a document is to that point. The value to search for must be an object holding the following parameters (or, at least, those parameters which are marked as required):
  - "center": Required. The coordinate of the center point for the query; expressed either as a (longitude, latitude) pair, or as an object with "lon" and "lat" properties.
  - "max_range": Optional. The maximum distance, in meters, that a document's coordinate may be from the center for that document to match the query. Defaults to an unlimited distance.
- "text": searches for a piece of text in a text field. The value to search for may be a single string, or an object holding the following parameters:
  - "text": <text to search for. If empty or missing, this query will match no results>
  - "op": <The operator to use when searching. One of "or", "and", "phrase", "near". Default=phrase>
  - "window": <only relevant if op is "phrase" or "near". window size in words; null=length of text. Integer or null. Default=null>
- "parse": parses a query, and searches for the query in a text field. The value to search for may be a single string, or an object holding the following parameters:
  - "text": <text to search for. If empty or missing, this query will match no results>
  - "op": <The default operator to use when searching. One of "or", "and". Default="and">
- "exists": used on the meta field (by default, named *_meta*) to search for documents in which a field exists. The value to search must be an array of values, each of which is either a fieldname to search for existence of, or "null" to search for existence of any field. Note that the ID and type special fields are excluded from the meta field, so it is not possible to search for their existence.

- "nonempty": used on the meta field (by default, named *_meta*) to search for documents in which a field exists and has a non-empty value. The value to search must be an array of values, each of which is either a fieldname to search for non-empty values in, or "null" to search for non-empty values in any field. Note that the ID and type special fields are excluded from the meta field, so it is not possible to search for nonempty values in them.
- "empty": used on the meta field (by default, named *_meta*) to search for documents in which a field exists and has an empty value. The value to search must be an array of values, each of which is either a fieldname to search for empty values in, or "null" to search for empty values in any field. Note that the ID and type special fields are excluded from the meta field, so it is not possible to search for empty values in them.
- "error": used on the meta field (by default, named *_meta*) to search for documents in which a field caused an error when processing. The value to search must be an array of values, each of which is either a fieldname to search for error values in, or "null" to search for error values in any field. Note that the ID and type special fields are excluded from the meta field, so it is not possible to search for error values in them.

8.1.3 Filtering results from another query

The results from the primary query are returned, filtered so that only those results which also match the filter are returned.

```
QUERY = {
    "query": QUERY, <optional - defaults to matchall>
    "filter": QUERY
}
```

8.2 Combining Queries

```
QUERY = {
    "and": [QUERY, ...]
}

QUERY = {
    "or": [QUERY, ...]
}

QUERY = {
    "xor": [QUERY, ...]
}

QUERY = {
    "and_not": [QUERY, ...]
}

QUERY = {
    "and_maybe": [QUERY, ...]
}
```

8.3 Scale the weights returned by a query.

Weights of a query, at any point in the tree, can be scaled by multiplying them by a constant factor.

```
QUERY = {
    "scale": {
         "query": QUERY,
         "factor": <multiplier to apply to the weight.  Double, >= 0. Required.>
    }
}
```

8.4 Getting additional information

8.4.1 Get co-occurrence counts for words in matching documents

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

Returns 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.

```
INFO = {
    "cooccur": {
        "prefix": <prefix of terms to check cooccurrence for>,
        "doc_limit": <number of matching documents to stop checking after.  null=unlimited.  Integer
        "result_limit": <number of term pairs to return results for.  null=unlimited.  Integer or nul
        "get_termfreqs": <set to true to also get frequencies of terms in the db.  Boolean.  Default=
        "stopwords": <list of stopwords - term suffixes to ignore.  Array of strings.  Default=[]>
    }
}
```

8.4.2 Getting term occurrence counts for words in matching documents

Warning - fairly slow.

Returns 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.

```
INFO = {
    "occur": {
        "prefix": <prefix of terms to check occurrence for>,
        "doc_limit": <number of matching documents to stop checking after.  null=unlimited.  Integer
        "result_limit": <number of terms to return results for.  null=unlimited.  Integer or null. De
        "get_termfreqs": <set to true to also get frequencies of terms in the db.  Boolean.  Default=
        "stopwords": <list of stopwords - term suffixes to ignore.  Array of strings.  Default=[]>
    }
}
```

8.5 Setting custom sort orders

By default, search results are ordered by a relevance score, calculated using the BM25 weighting scheme. The internal RestPose architecture allows for considerable flexibility in how weights are calculated, and also allows for ordering

by schemes other than relevance score (eg, by a field value). As yet, little of this flexibility is exposed in the API, but more is planned to be. Contact the author if you wish particular options to be made available.

Currently, the sort order can be set using the `order_by` configuration. A sort order may be set using a field, as follows:

```
ORDER_BY = [
    {"field": <field name>,
     "ascending": ASCENDING  // Optional - defaults to true
    }
]

ASCENDING = <boolean - if true, the first results returned (ie, lowest rank) will have the lowest val
```

Alternately, the sort order can be set to be relevance order (which is the default order):

```
ORDER_BY = [
    {"score": "weight",
     "ascending": false  // Optional - defaults to false.  true is not allowed, but this is included
    }
]
```

At present, the list of sort orders may only contain exactly one item.

8.6 Specifying a result set offset relative to a document

Sometimes, it's useful to be able to get the section of a search result set containing a particular document, rather than just a section of a result set based on an offset. For example, imagine you're providing an interface which allows users to page through a resultset where documents are constantly being added to the underlying database, so additional documents may be added at any time.

In this situation, it may be better to ask for the results which follow a particular document, rather than to ask for the next page of results by a fixed offset. This is particularly true if documents are being returned sorted by most-recent first; using this technique.

Note that this interface can also be used to get the rank of a given document in a set of search results, without having to iterate through the search results on the client side.

To get a set of search results based on the position of a document in the search results, you can add a "fromdoc" property to the search:

```
FROMDOC = {
    "type": <string: the document type of the base document>
    "id": <string: the document ID of the base document>
    "from": <integer: offset relative to the base document - may be negative.  Defaults to 0>
    "pagesize": <integer: number of results to calculate in each page when looking for the base docur
}
```

Note that if a "fromdoc" property is supplied for a search, the "from" property must be 0 (or absent).

8.7 Search results

Search results are returned as a JSON object, with the following properties.

- `from`: (int) The *from* value used when performing the search.
- `size_requested`: (int) The *size* value used when performing the search.
- `check_at_least`: (int) The *check_at_least* value used when performing the search.
- `total_docs`: (int) The total number of documents searched through.
- `matches_lower_bound`: (int) A lower bound on the number of matching documents. This will be precise if *check_at_least* was -1, or was high enough to ensure that all matches were checked.
- `matches_estimated`: (int) An estimate on the number of matching documents. This will be precise if *check_at_least* was -1, or was high enough to ensure that all matches were checked.
- `matches_upper_bound`: (int) An upper bound on the number of matching documents. This will be precise if *check_at_least* was -1, or was high enough to ensure that all matches were checked.
- `items`: (array) An array of results from searching. Each result is a object, keyed by fieldname, holding the stored fields for that result. The search may limit which fields are returned.

CHAPTER

NINE

CLIENTS

While you can access RestPose directly over HTTP from any language and environment which supports making HTTP requests, it will often be more convenient to use a dedicated client, to provide tighter and smoother integration into your chosen lanugae.

This section lists the languages for which clients are currently available:

- Python. http://github.com/rboulton/restpose-py/ This is an "official" client, maintained by the RestPose team. It is released whenever a new version of RestPose server is, and the latest version should always provide access to all features of RestPose Server.

If you know of, or implement, a client which isn't already in this list, please let us know (ideally in the form of a pull request to update this documentation request).

CHAPTER

TEN

INDICES AND TABLES

- *genindex*
- *search*

HTTP ROUTING TABLE

/

GET /, **??**

/coll

GET /coll, **??**
GET /coll/(collection_name)/taxonomy,
??
GET /coll/(collection_name)/checkpoint/(checkpoint_id),
??
GET /coll/(collection_name)/type/(type)/search,
??
GET /coll/(collection_name)/config, **??**
GET /coll/(collection_name)/taxonomy/(taxonomy_name),
??
GET /coll/(collection_name)/checkpoint,
??
GET /coll/(collection_name)/search, **??**
GET /coll/(collection_name), **??**
GET /coll/(collection_name)/taxonomy/(taxonomy_name)/top,
??
GET /coll/(collection_name)/type/(type)/id/(id),
??
GET /coll/(collection_name)/taxonomy/(taxonomy_name)/id/(cat_id)/parent/(parent_id),
??
GET /coll/(collection_name)/taxonomy/(taxonomy_name)/id/(cat_id),
??
PUT /coll/(collection_name)/config, **??**
PUT /coll/(collection_name)/type/(type)/id/(id),
??
PUT /coll/(collection_name)/taxonomy/(taxonomy_name)/id/(cat_id)/parent/(parent_id),
??
PUT /coll/(collection_name)/taxonomy/(taxonomy_name)/id/(cat_id),
??
POST /coll/(collection_name)/id/(id),
??
POST /coll/(collection_name)/type/(type)/search,
??
POST /coll/(collection_name)/checkpoint,
??
POST /coll/(collection_name)/type/(type),
??
POST /coll/(collection_name)/search, **??**
POST /coll/(collection_name), **??**
DELETE /coll/(collection_name)/taxonomy/(taxonomy_n
??
DELETE /coll/(collection_name)/taxonomy/(taxonomy_n
??
DELETE /coll/(collection_name)/type/(type)/id/(id),
??
DELETE /coll/(collection_name), **??**
DELETE /coll/(collection_name)/taxonomy/(taxonomy_n
??

/static

GET /static/(static_path), **??**

/status

GET /status, **??**