Scout Documentation

Release 3.0.3

Charles Leifer

Feb 21, 2020

Contents

scout

sqlite search engine

scout is a RESTful search server written in Python with a focus on using lightweight components:

- search powered by sqlite's full-text search extension
- database access coordinated using peewee ORM
- web application built with flask framework

Scout aims to be a lightweight, RESTful search server in the spirit of [ElasticSearch](https://www.elastic.co), powered by the SQLite full-text search extension. In addition to search, Scout can be used as a document database, supporting complex filtering operations. Arbitrary files can be attached to documents and downloaded through the REST API.

Scout is simple to use, simple to deploy and *just works*.

CHAPTER 1

Features

- multiple search indexes present in a single database.
- restful design for easy indexing and searching.
- simple key-based authentication (optional).
- lightweight, low resource utilization, minimal setup required.
- store search content and arbitrary metadata.
- attach files or BLOBs to indexed documents.
- multiple result ranking algorithms, porter stemmer.
- besides full-text search, perform complex filtering based on metadata values.
- comprehensive unit-tests.
- supports SQLite FTS4.

named in honor of the best dog ever,

CHAPTER 2

Table of contents

Contents:

2.1 Installing and Testing

Most users will want to simply install the latest version, hosted on PyPI:

```
pip install scout
```

2.1.1 Installing with git

The project is hosted at https://github.com/coleifer/scout and can be installed using git:

```
git clone https://github.com/coleifer/scout.git
cd scout
python setup.py install
```

Note: On some systems you may need to use `sudo python setup.py install` to install scout system-wide.

2.1.2 Dependencies

Scout has the following Python dependencies:

- Flask
- Peewee

If you installed Scout using `pip` then the dependencies will have automatically been installed for you. Otherwise be sure to install `flask` and `peewee`.

Scout also depends on SQLite and the SQLite full-text search extension. SQLite is installed by default on most operating systems, and is generally compiled with FTS, so typically no additional installation is necessary.

If you wish, you can also run Scout using the gevent WSGI server. This process is described in the *Hacks* document.

2.1.3 Running tests

You can test your installation by running the test suite.

```
python tests.py
```

2.2 Scout Server

Scout server is a RESTful Flask application that provides endpoints for managing indexes, documents, metadata and performing searches. To get started with Scout, you can simply run the server from the command-line and specify the path to a SQLite database file which will act as the search index.

Note: If the database does not exist, it will be created.

```
$ python scout.py my_search_index.db
 * Running on http://127.0.0.1:8000/ (Press CTRL+C to quit)
```

This will run Scout locally on port 8000 using the Werkzeug multi-threaded WSGI server. The werkzeug server is perfect for getting started and small deployments. You can find examples of using high performance WSGI servers in the *deployment section*.

2.2.1 API Endpoints

There are four main concepts in Scout:

- Indexes
- Documents
- Attachments
- Metadata

Indexes have a name and may contain any number of documents.

Documents have content, which is indexed for search, and may be associated with any number of indexes.

Attachments are arbitrary files which are associated with a document. For instance, if you were using Scout to provide search over a library of PDFs, your *Document* might contain the key search terms from the PDF and the actual PDF would be linked to the document as an attachment. A document may have any number of attachments, or none at all.

Documents also can have *metadata*, arbitrary key/value pairs. Besides full-text search, Scout allows complex filtering based on metadata values. So in addition to storing useful things alongside your documents, you can also use metadata to provide an additional layer of filtering.

2.2.2 Index list: "/"

The index list endpoint returns the list of indexes and the number of documents contained within each. The list is not paginated and will display all available indexes. New indexes can be created by POST-ing a name to this URL.

Valid GET parameters:

- `page`: which page of results to fetch, by default 1.
- `ordering`: order in which to return the indexes. By default they are returned ordered by name. Valid values are `name`, `id`, and `document_count`. By prefixing the name with a *minus* sign ("-") you can indicate the results should be ordered descending.

Example GET request and response:

```
$ curl localhost:8000/
```

Response:

```
{
  "indexes": [
    {
      "document_count": 75,
      "documents": "/blog/",
      "id": 1,
      "name": "blog"
    },
    {
      "document_count": 36,
      "documents": "/photos/",
      "id": 2,
      "name": "photos"
    }
  ],
  "ordering": [],
  "page": 1,
  "pages": 1
}
```

Example POST request and response:

```
$ curl -H "Content-Type: application/json" -d '{"name": "test-index"}' localhost:8000/
```

Response:

```
{
  "document_count": 0,
  "documents": [],
  "id": 3,
  "name": "test-index",
  "page": 1,
  "pages": 0
}
```

The POST response corresponds to the serialized index detail for the newly-created index.

2.2.3 Index detail: "/:index-name/"

The index detail returns the name and ID of the index, as well as a paginated list of documents associated with the index. The index can be re-named by POSTing a `name` to this URL.

Valid GET parameters:

- `q`: full-text search query.
- `page`: which page of results to fetch, by default 1.
- `ordering`: order in which to return the documents. By default they are returned in arbitrary order, unless a search query is present, in which case they are ordered by relevance. Valid choices are `id`, `identifier`, `content`, and `score`. By prefixing the name with a *minus* sign ("-") you can indicate the results should be ordered descending. **Note**: this parameter can appear multiple times.
- `ranking`: when a full-text search query is specified, this parameter determines the ranking algorithm. Valid choices are:
  - `bm25`: use the Okapi BM25 algorithm. This is only available if your version of SQLite supports FTS4 or FTS5.
  - `simple`: use a simple, efficient ranking algorithm.
  - `none`: do not use any ranking algorithm. Search results will not have a *score* attribute.
- **Arbitrary metadata filters**. See *Filtering on Metadata* for a description of metadata filtering..

When a search query is present, each returned document will have an additional field named `score`. This field contains the numerical value the scoring algorithm gave to the document. To disable scores when searching, you can specify `ranking=none`.

Example `GET` request and response.

```
$ curl localhost:8000/test-index/?q=test
```

Response:

```
{
  "document_count": 3,
  "documents": [
    {
      "attachments": [],
      "content": "test charlie document",
      "id": 115,
      "identifier": null,
      "indexes": [
        "test-index"
      ],
      "metadata": {
        "is_kitty": "no"
      },
      "score": -0.022727272727272728
    },
    {
      "attachments": [
        {
          "data": "/documents/116/attachments/example.jpg/download/",
          "data_length": 31337,
          "filename": "example.jpg",
          "mimetype": "image/jpeg",
```

(continues on next page)

(continued from previous page)


```
          "timestamp: "2016-01-04 13:37:00"
        }
      ],
      "content": "test huey document",
      "id": 116,
      "identifier": null,
      "indexes": [
        "test-index"
      ],
      "metadata": {
        "is_kitty": "yes"
      },
      "score": -0.022727272727272728
    },
    {
      "attachments": [],
      "content": "test mickey document",
      "id": 117,
      "identifier": null,
      "indexes": [
        "test-index"
      ],
      "metadata": {
        "is_kitty": "no"
      },
      "score": -0.022727272727272728
    }
  ],
  "filtered_count": 3,
  "filters": {},
  "id": 3,
  "name": "test-index",
  "ordering": [],
  "page": 1,
  "pages": 1,
  "ranking": "bm25",
  "search_term": "test"
}
```


`POST` requests update the `name` of the index, and like the *index_list* view, accept a `name` parameter. For example request and response, see the above section on creating a new index.

`DELETE` requests will delete the index, but all documents will be preserved in the database.

Example of deleting an index:

```
$ curl -X DELETE localhost:8000/photos/
```

Response:


```
{"success": true}
```


2.2.4 Filtering on Metadata

Suppose we have an index that contains all of our contacts. The search content consists of the person's name, address, city, and state. We also have stored quite a bit of metadata about each person. A person record might look like this:

```
{'content': "Huey Leifer 123 Main Street Lawrence KS 66044"}
```

The metadata for this record consists of the following:

```
{'metadata': {
  'dob': '2010-06-01',
  'city': 'Lawrence',
  'state': 'KS',
}}
```

To search for all my relatives living in Kansas, I could use the following URL:

`/contacts-index/?q=Leifer+OR+Morgan&state=KS`

Let's say we want to search our contacts index for all people who were born in 1983. We could use the following URL:

`/contacts-index/?dob__ge=1983-01-01&dob__lt=1984-01-01`

To search for all people who live in Lawrence or Topeka, KS we could use the following URL:

`/contacts-index/?city__in=Lawrence,Topeka&state=KS`

Scout will take all filters and return only those records that match all of the given conditions. However, when the same key is used multiple times, Scout will use `OR` to join those clauses. For example, another way we could query for people who live in Lawrence or Topeka would be:

`/contacts-index/search/?q=*&city=Lawrence&city=Topeka&state=KS`

As you can see, we're querying `city=XXX` twice. Scout will interpret that as meaning `(city=Lawrence OR city=Topeka) AND state=KS`.

Query operations

There are a number of operations available for use when querying metadata. Here is the complete list:

- `keyname__eq`: Default (when only the key name is supplied). Returns documents whose metadata contains the given key/value pair.
- `keyname__ne`: Not equals.
- `keyname__ge`: Greater-than or equal-to.
- `keyname__gt`: Greater-than.
- `keyname__le`: Less-than or equal-to.
- `keyname__lt`: Less-than.
- `keyname__in`: In. The value should be a comma-separated list of values to match.
- `keyname__contains`: Substring search.
- `keyname__startswith`: Prefix search.
- `keyname__endswith`: Suffix search.
- `keyname__regex`: Search using a regular expression.

2.2.5 Document list: "/documents/"

The document list endpoint returns a paginated list of all documents, regardless of index. New documents are created by `POST`-ing the content, index(es) and optional metadata.

Valid GET parameters:

- `q`: full-text search query.
- `page`: which page of documents to fetch, by default 1.
- `index`: the name of an index to restrict the results to. **Note**: this parameter can appear multiple times.
- `ordering`: order in which to return the documents. By default they are returned in arbitrary order, unless a search query is present, in which case they are ordered by relevance. Valid choices are `id`, `identifier`, `content`, and `score`. By prefixing the name with a *minus* sign ("-") you can indicate the results should be ordered descending. **Note**: this parameter can appear multiple times.
- `ranking`: when a full-text search query is specified, this parameter determines the ranking algorithm. Valid choices are:
  - `bm25`: use the Okapi BM25 algorithm. This is only available if your version of SQLite supports FTS4 or FTS5.
  - `simple`: use a simple, efficient ranking algorithm.
  - `none`: do not use any ranking algorithm. Search results will not have a *score* attribute.
- **Arbitrary metadata filters**. See *Filtering on Metadata* for a description of metadata filtering..

When a search query is present, each returned document will have an additional field named `score`. This field contains the numerical value the scoring algorithm gave to the document. To disable scores when searching, you can specify `ranking=none`.

Example `GET` request and response. In the request below we are searching for the string *"test"* in the `photos`, `articles` and `videos` indexes.

```
$ curl localhost:8000/documents/?q=test&index=photos&index=articles&index=videos
```

Response:

```
{
  "document_count": 207,
  "documents": [
    {
      "attachments": [
        {
          "data": "/documents/72/attachments/example.jpg/download/",
          "data_length": 31337,
          "filename": "example.jpg",
          "mimetype": "image/jpeg",
          "timestamp: "2016-03-01 13:37:00"
        }
      ],
      "content": "test photo",
      "id": 72,
      "identifier": null,
      "indexes": [
        "photos"
      ],
      "metadata": {
```

(continues on next page)

(continued from previous page)


```
        "timestamp": "2016-03-01 13:37:00"
      },
      "score": -0.01304
    },
    {
      "attachments": [
        {
          "data": "/documents/61/attachments/movie.mp4/download/",
          "data_length": 3131337,
          "filename": "movie.mp4",
          "mimetype": "video/mp4",
          "timestamp: "2016-03-02 13:37:00"
        }
      ],
      "content": "test video upload",
      "id": 61,
      "identifier": null,
      "indexes": [
        "videos"
      ],
      "metadata": {
        "timestamp": "2016-03-02 13:37:00"
      },
      "score": -0.01407
    }
  ],
  "filtered_count": 2,
  "filters": {},
  "ordering": [],
  "page": 1,
  "pages": 1,
  "ranking": "bm25",
  "search_term": "test"
}
```


`POST` requests should have the following parameters:

- `content` (required): the document content.
- `index` or `indexes` (required): the name(s) of the index(es) the document should be associated with.
- `identifier` (optional): an application-defined identifier for the document.
- `metadata` (optional): arbitrary key/value pairs.

Example `POST` request creating a new document:

```
$ curl \
    -H "Content-Type: application/json" \
    -d '{"content": "New document", "indexes": ["test-index"]}' \
    http://localhost:8000/documents/
```

Response on creating a new document:


```
{
  "content": "New document",
  "id": 121,
  "indexes": [
```


(continues on next page)

(continued from previous page)

```
    "test-index"
  ],
  "metadata": {}
}
```

2.2.6 Document detail: "/documents/:document-id/"

The document detail endpoint returns document content, indexes, and metadata. Documents can be updated or deleted by using `POST` and `DELETE` requests, respectively. When updating a document, you can update the `content`, `index(es)`, and/or `metadata`.

Warning: If you choose to update metadata, all current metadata for the document will be removed, so it's really more of a "replace" than an "update".

Example `GET` request and response:

```
$ curl localhost:8000/documents/118/
```

Response:

```
{
  "attachments": [],
  "content": "test zaizee document",
  "id": 118,
  "identifier": null,
  "indexes": [
    "test-index"
  ],
  "metadata": {
    "is_kitty": "yes"
  }
}
```

Here is an example of updating the content and indexes using a `POST` request:

```
$ curl \
    -H "Content-Type: application/json" \
    -d '{"content": "test zaizee updated", "indexes": ["test-index", "blog"]}' \
    http://localhost:8000/documents/118/
```

Response:

```
{
  "content": "test zaizee updated",
  "id": 118,
  "indexes": [
    "blog",
    "test-index"
  ],
  "metadata": {
    "is_kitty": "yes"
  }
}
```

DELETE requests can be used to completely remove a document.

Example DELETE request and response:

```
$ curl -X DELETE localhost:8000/documents/121/
```

Response:

```
{"success": true}
```

2.2.7 Attachment list: "/documents/:document-id/attachments/"

The attachment list endpoint returns a paginated list of all attachments associated with a given document. New attachments are created by POST-ing a file to this endpoint.

Valid GET parameters:

- page: which page of attachments to fetch, by default 1.
- ordering: order in which to return the attachments. By default they are returned by filename. Valid choices are id, hash, filename, mimetype, and timestamp. By prefixing the name with a *minus* sign ("-") you can indicate the results should be ordered descending. **Note**: this parameter can appear multiple times.

Example GET request and response.

```
$ curl localhost:8000/documents/13/attachments/?ordering=timestamp
```

Response:

```
{
  "attachments": [
    {
      "data": "/documents/13/attachments/banner.jpg/download/",
      "data_length": 135350,
      "document": "/documents/13/",
      "filename": "banner.jpg",
      "mimetype": "image/jpeg",
      "timestamp": "2016-03-01 13:37:01"
    },
    {
      "data": "/documents/13/attachments/background.jpg/download/",
      "data_length": 25039,
      "document": "/documents/13/",
      "filename": "background.jpg",
      "mimetype": "image/jpeg",
      "timestamp": "2016-03-01 13:37:02"
    }
  ],
  "ordering": ["timestamp"],
  "page": 1,
  "pages": 1
}
```

POST requests should contain the attachments as form-encoded files. The *Scout client* will handle this automatically for you.

Example POST request uploading a new attachment:

```
$ curl \
    -H "Content-Type: multipart/form-data" \
    -F 'data=""' \
    -F "file_0=@/path/to/image.jpg" \
    -X POST \
    http://localhost:8000/documents/13/attachments/
```

Response on creating a new attachment:

```
{
  "attachments": [
    {
      "data": "/documents/13/attachments/some-image.jpg/download/",
      "data_length": 18912,
      "document": "/documents/13/",
      "filename": "some-image.jpg",
      "mimetype": "image/jpeg",
      "timestamp": "2016-03-14 13:38:00"
    }
  ]
}
```

Note: You can upload multiple attachments at the same time.

2.2.8 Attachment detail: "/documents/:document-id/attachments/:filename/"

The attachment detail endpoint returns basic information about the attachment, as well as a link to download the actual attached file. Attachments can be updated or deleted by using `POST` and `DELETE` requests, respectively. When you update an attachment, the original is deleted and a new attachment created for the uploaded content.

Example `GET` request and response:

```
$ curl localhost:8000/documents/13/attachments/test-image.png/
```

Response:

```
{
  "data": "/documents/13/attachments/test-image.png/download/",
  "data_length": 3710133,
  "document": "/documents/13/",
  "filename": "test-image.png",
  "mimetype": "image/png",
  "timestamp": "2016-03-14 22:10:00"
}
```

`DELETE` requests are used to **detach** a file from a document.

Example `DELETE` request and response:

```
$ curl -X DELETE localhost:8000/documents/13/attachments/test-image.png/
```

Response:

```
{"success": true}
```

2.2.9 Attachment download: "/documents/:document-id/attachments/:filename/download/"

The attachment download endpoint is a special URL that returns the attached file as a downloadable HTTP response. This is the only way to access an attachment's underlying file data.

To download an attachment, simply send a `GET` request to the attachment's "data" URL:

```
$ curl http://localhost:8000/documents/13/attachments/banner.jpg/download/
```

2.2.10 Example of using Authentication

Scout provides very basic key-based authentication. You can specify a single, global key which must be specified in order to access the API.

To specify the API key, you can pass it in on the command-line or specify it in a configuration file (described below).

Example of running scout with an API key:

```
$ python scout.py -k secret /path/to/search.db
```

If we try to access the API without specifying the key, we get a `401` response stating Invalid API key:

```
$ curl localhost:8000/
Invalid API key
```

We can specify the key as a header:

```
$ curl -H "key: secret" localhost:8000/
{
  "indexes": []
}
```

Alternatively, the key can be specified as a `GET` argument:

```
$ curl localhost:8000/?key=secret
{
  "indexes": []
}
```

2.2.11 Configuration and Command-Line Options

The easiest way to run Scout is to invoke it directly from the command-line, passing the database in as the last argument:

```
$ python scout.py /path/to/search.db
```

The database file can also be specified using the SCOUT_DATABASE environment variable:

```
$ SCOUT_DATABASE=/path/to/search.db python scout.py
```

Scout supports a handful of configuration options to control it's behavior when run from the command-line. The following table describes these options:

- `-H`, `--host`: set the hostname to listen on. Defaults to `127.0.0.1`
- `-p`, `--port`: set the port to listen on. Defaults to `8000`.

- `-u`, `--url-prefix`: url path to prefix Scout API with, e.g. "/search".
- `-s`, `--stem`: set the stemming algorithm. Valid options are `simple` and `porter`. Defaults to `porter` stemmer. This option only will be in effect when a new database is created, as the stemming algorithm is part of the table definition.
- `-d`, `--debug`: boolean flag to run Scout in debug mode.
- `-c`, `--config`: set the configuration file (a Python module). See the configuration options for available settings.
- `--paginate-by`: set the number of documents displayed per page of results. Default is 50.
- `-k`, `--api-key`: set the API key required to access Scout. By default no authentication is required.
- `-C`, `--cache-size`: set the size of the SQLite page cache (in MB), defaults to 64.
- `-f`, `--fsync`: require fsync after every SQLite transaction is committed.
- `-j`, `--journal-mode`: specify SQLite journal-mode. Default is "wal".
- `-l`, `--logfile`: configure file for log output.

2.2.12 Python Configuration File

For more control, you can override certain settings and configuration values by specifying them in a Python module to use as a configuration file.

The following options can be overridden:

- `AUTHENTICATION` (same as `-k` or `--api-key`).
- `DATABASE`, the path to the SQLite database file containing the search index. This file will be created if it does not exist.
- `DEBUG` (same as `-d` or `--debug`).
- `HOST` (same as `-H` or `--host`).
- `PAGINATE_BY` (same as `--paginate-by`).
- `PORT` (same as `-p` or `--port`).
- `SECRET_KEY`, which is used internally by Flask to encrypt client-side session data stored in cookies.
- `STEM` (same as `-s` or `--stem`).

Note: Options specified on the command-line will override any options specified in the configuration file.

Example configuration file:

```
# search_config.py
AUTHENTICATION = 'my-secret-key'
DATABASE = 'my_search.db'
HOST = '0.0.0.0'
PORT = 1234
STEM = 'porter'
```

Example of running Scout with the above config file. Note that since we specified the database in the config file, we do not need to pass one in on the command-line.

```
$ python scout.py -c search_config.py
```

You can also specify the configuration file using the `SCOUT_CONFIG` environment variable:

```
$ SCOUT_CONFIG=search_config.py python scout.py
```

2.3 Scout Client

Scout comes with a simple Python client. This document describes the client API.

class Scout (*endpoint*[, *key=None*])

The *Scout* class provides a simple, Pythonic API for interacting with and querying a Scout server.

Parameters

- **endpoint** – The base URL the Scout server is running on.
- **key** – The authentication key (if used) required to access the Scout server.

Example of initializing the client:

```
>>> from scout_client import Scout
>>> scout = Scout('https://search.my-site.com/', key='secret!')
```

get_indexes (***kwargs*)

Return the list of indexes available on the server.

See *Index list: "/"* for more information.

create_index (*name*)

Create a new index with the given name. If an index with that name already exists, you will receive a 400 response.

See the POST section of *Index list: "/"* for more information.

rename_index (*old_name*, *new_name*)

Rename an existing index.

delete_index (*name*)

Delete an existing index. Any documents associated with the index will **not** be deleted.

get_index (*name*, ***kwargs*)

Return the details about the particular index, along with a paginated list of all documents stored in the given index.

The following optional parameters are supported:

Parameters

- **q** – full-text search query to be run over the documents in this index.
- **ordering** – columns to sort results by. By default, when you perform a search the results will be ordered by relevance.
- **ranking** – ranking algorithm to use. By default this is `bm25`, however you can specify `simple` or `none`.
- **page** – page number of results to retrieve
- ****filters** – Arbitrary key/value pairs used to filter the metadata.

The *Filtering on Metadata* section describes how to use key/value pairs t construct filters on the document's metadata.

See *Index detail: "/:index-name/"* for more information.

create_document(*content*, *indexes*[, *identifier=None*[, *attachments=None*[, ***metadata*]]])
Store a document in the specified index(es).

Parameters

- **content** (*str*) – Text content to expose for search.
- **indexes** – Either the name of an index or a list of index names.
- **identifier** – Optional alternative user-defined identifier for document.
- **attachments** – An optional mapping of filename to file-like object, which should be uploaded and stored as attachments on the given document.
- **metadata** – Arbitrary key/value pairs to store alongside the document content.

update_document([*document_id=None*[, *content=None*[, *indexes=None*[, *metadata=None*[, *identifier=None*[, *attachments=None*]]]]]])
Update one or more attributes of a document that's stored in the database.

Parameters

- **document_id** (*int*) – The integer document ID (required).
- **content** (*str*) – Text content to expose for search (optional).
- **indexes** – Either the name of an index or a list of index names (optional).
- **metadata** – Arbitrary key/value pairs to store alongside the document content (optional).
- **identifier** – Optional alternative user-defined identifier for document.
- **attachments** – An optional mapping of filename to file-like object, which should be uploaded and stored as attachments on the given document. If a filename already exists, it will be over-written with the new attachment.

Note: If you specify metadata when updating a document, existing metadata will be replaced by the new metadata. To simply clear out the metadata for an existing document, pass an empty `dict`.

---

delete_document(*document_id*)
Remove a document from the database, as well as all indexes.

Parameters **document_id** (*int*) – The integer document ID.

get_document(*document_id*)
Retrieve content for the given document.

Parameters **document_id** (*int*) – The integer document ID.

get_documents(***kwargs*)
Retrieve a paginated list of all documents in the database, regardless of index. This method can also be used to perform full-text search queries across the entire database of documents, or a subset of indexes.

The following optional parameters are supported:

Parameters

- **q** – full-text search query to be run over the documents in this index.

- **ordering** – columns to sort results by. By default, when you perform a search the results will be ordered by relevance.
- **index** – one or more index names to restrict the results to.
- **ranking** – ranking algorithm to use. By default this is `bm25`, however you can specify `simple` or `none`.
- **page** – page number of results to retrieve
- ****filters** – Arbitrary key/value pairs used to filter the metadata.

The *Filtering on Metadata* section describes how to use key/value pairs t construct filters on the document's metadata.

See *Document list: "/documents/"* for more information.

attach_files(*document_id*, *attachments*)

Parameters

- **document_id** – The integer ID of the document.
- **attachments** – A dictionary mapping filename to file-like object.

Upload the attachments and associate them with the given document.

For more information, see *Attachment list: "/documents/:document-id/attachments/"*.

detach_file(*document_id*, *filename*)

Parameters

- **document_id** – The integer ID of the document.
- **filename** – The filename of the attachment to remove.

Detach the specified file from the document.

update_file(*document_id*, *filename*, *file_object*)

Parameters

- **document_id** – The integer ID of the document.
- **filename** – The filename of the attachment to update.
- **file_object** – A file-like object.

Replace the contents of the current attachment with the contents of `file_object`.

get_attachments(*document_id*, ***kwargs*)

Retrieve a paginated list of attachments associated with the given document.

The following optional parameters are supported:

Parameters

- **ordering** – columns to use when sorting attachments.
- **page** – page number of results to retrieve

For more information, see *Attachment list: "/documents/:document-id/attachments/"*.

get_attachment(*document_id*, *filename*)

Retrieve data about the given attachment.

For more information, see *Attachment detail: "/documents/:document-id/attachments/:filename/"*.

download_attachment(*document_id*, *filename*)
Download the specified attachment.

For more information, see *Attachment download: "/documents/:document-id/attachments/:filename/download/"*.

2.4 Deployment

When scout is run from the command-line, it will use the multi-threaded Werkzeug WSGI server. While this server is perfect for development and small installations, you may want to use a high-performance WSGI server to deploy Scout.

Scout provides a WSGI app, so you can use any WSGI server for deployment. Popular choices are:

- Gevent
- Gunicorn
- uWSGI

The Flask documentation also provides a list of popular WSGI servers and how to integrate them with Flask apps. Since Scout is a Flask application, all of these examples should work with minimal modification:

http://flask.pocoo.org/docs/0.10/deploying/wsgi-standalone/

2.4.1 Gevent

Here is an example wrapper script for running Scout using the Gevent WSGI server:

```
from gevent import monkey
monkey.patch_all()

from gevent.pywsgi import WSGIServer
from scout.server import parse_options

# Parse command-line options and return a Flask app.
app = parse_options()

# Run the WSGI server on localhost:8000.
WSGIServer(('127.0.0.1', 8000), app).serve_forever()
```

You could then run the wrapper script using a tool like supervisord or another process manager.

2.4.2 Gunicorn

Here is an example wrapper script for running Scout using Gunicorn.

```
# Wrapper script to initialize database.
from scout.server import parse_options
app = parse_options()
```

Here is how to run gunicorn using the above wrapper script:

```
$ gunicorn --workers=4 --bind=127.0.0.1:8000 --worker-class=gevent wrapper:app
```

2.4.3 uWSGI

Here is an example wrapper script for uWSGI.

```
# Wrapper script to initialize database.
from scout import parse_options
app = parse_options()
```

Here is how you might run using the above wrapper script:

```
$ uwsgi --http :8000 --wsgi-file wrapper.py --master --processes 4 --threads 2
```

It is common to run uWSGI behind Nginx. For more information check out the uWSGI docs.

2.5 Hacks

In this document you will find some of the hacks users of Scout have come up with to do novel things.

Most of these techniques involve wrapping the Scout server application with an additional module. Since Scout server is a normal Python module, and the WSGI app is just an object within that module, there is no magic needed to extend the behavior of Scout.

2.5.1 Adding CORS headers

I wanted to be able to query my Scout index from JavaScript running on a different host. To get this working I needed to add some special headers to each response from the API (more info on CORS).

To accomplish this I created a wrapper module that wraps the Scout server Flask app and implements a special `after_request` hook:

Here is the wrapper module:

```
from scout.server import parse_options

app = parse_options()

@app.after_request
def add_cors_header(response):
    response.headers['Access-Control-Allow-Origin'] = 'http://myhost.com'
    response.headers['Access-Control-Allow-Headers'] = 'key,Content-Type'
    response.headers['Access-Control-Allow-Methods'] = 'GET,POST,DELETE'
    return response
```

2.5.2 Adding logging

To log exceptions within the Scout server, I created a wrapper for the Scout server WSGI app and added a handler to Flask's built-in app logger.

Here is the wrapper code:

```
import logging
import os

from scout.server import parse_options

app = parse_options()

cur_dir = os.path.realpath(os.path.dirname(__file__))
log_dir = os.path.join(cur_dir, 'logs')

handler = logging.FileHandler(os.path.join(log_dir, 'scout-error.log'))
handler.setLevel(logging.ERROR)
app.logger.addHandler(handler)
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index