birding Documentation

Release 0.0

Parse.ly

December 02, 2015

Contents

birding is an open source project to produce a stream of recent twitter activity based on a sequence of search terms, using only twitter's public APIs. It serves as both a standalone project and a demo of distributed real-time computing with Python using Storm/streamparse and Kafka/pykafka.

Problem statement & topology describes the problem and how it fits into a topology. *Downloading and running birding* describes how to interact with birding for development, demo, or light usage. *A tour of birding's implementation* provides a light introduction to internals. *Using birding in production* discusses how birding is packaged for production use in an existing streamparse project. *Configuring birding* discusses various options for birding behavior when running locally or in production.

CHAPTER 1

Problem statement & topology

1.1 Problem Statement

Take as input a sequence of terms and timestamps and produce a "filtered firehose" of twitter activity using only twitter's public APIs, without requiring special API access to twitter or any third party.

1.2 Specifics

- Input is in the format of (term, timestamp), where *term* is any string and *timestamp* is a date/time value in an ISO 8601 format, e.g. `2015-06-25T08:00Z`.
- The motivating use-case:
  - provides this input as a Kafka topic
  - prefers output be sent to a Kafka topic & include full twitter API results
  - prefers the solution be implemented in Python

1.3 Observations

Twitter provides GET search/tweets to get relevant Tweets (status updates) matching a specified query. Any detail not provided in the search results can be accessed with GET statuses/lookup, looking up multiple status updates in a batched request.

The problem has potentially unbounded streams of data, which makes Storm a relevant technology for the solution. Given that the motivating use-case prefers Python with Kafka I/O, streamparse and pykafka are relavant.

1.4 Topology

Given the problem statement, a streaming solution looks something like:

1.5 Other Goals

The solution should:

- Encode best practices about how to use Storm/streamparse and Kafka/pykafka.
- Be fully public & open source to serve as an example project, so it should not depend on anything specific to a specific company/organization. Depending on the publicly scrutable Twitter API is, of course, okay.
- Include basic command-line tools for testing the topology with data and ways to configure things like Twitter authentication credentials.

Next, goto one of:

- *Downloading and running birding*
- *A tour of birding's implementation*

CHAPTER 2

Downloading and running *birding*

Note: Existing streamparse projects should include the birding Python package instead of cloning the birding repository, which is described in *Using birding in production*.

The birding project fully automates dependencies for the purposes of development, demo, or light usage. In a terminal on a Unix-like system, clone the birding repository:

```
git clone https://github.com/Parsely/birding.git
cd birding
```

Then run:

```
make run
```

The birding project makes every effort to detect if an underlying dependency is unmet. If *make run* fails, look for messages indicating what is missing or what went wrong. If an error message says that an address is in use, look for other processes on the system which are currently using the referenced network port, then shut them down in order to run birding. If an error is unclear, submit an issue including a build log and mention your operating system. To create a *build.log*:

```
make run 2>&1 | tee build.log
```

When birding is running, its console output is verbose as it includes all output of zookeeper, kafka, storm, and streamparse. Note that – as with all streamparse projects – output from the birding code itself ends up in the `logs/` directory and not in the console. To stop running birding, issue a keyboard interrupt in the console with Control-C:

```
Control-C
```

Using *make run* will pick up *birding.yml* as the project configuration file if it exists in the root directory next to the *Makefile*. See *Configuring birding*. This simple *birding.yml* to sets the search terms used by birding:

```
TermCycleSpout:
  terms:
  - mocking bird
  - carrier pigeon
```

Data for the project ends up in a directory relative to the project root. Clean runtime data with:

```
make clean-data
```

Build docs with `make docs` and check for Python errors by static analysis with `make flakes`. Make allows multiple targets at once:

```
make clean-data flakes run
```

Next, goto one of:

- *A tour of birding's implementation*
- *Using birding in production*
- *Configuring birding*

CHAPTER 3

A tour of *birding*'s implementation

3.1 Python Twitter Client

There are many Python packages for Twitter. The Python Twitter Tools project (`pip install twitter`) is of interest because:

1. It has a command-line application to get twitter activity which includes a straightforward authentication workflow to log into twitter and get OAuth credentials, using a PIN-Based workflow.
2. It provides APIs in Python which bind to twitter's public APIs in a dynamic and predictable way, where Python attribute and method names translate to URL paths, e.g. `twitter.statuses.friends_timeline()` retrieves data from `http://twitter.com/statuses/friends_timeline.json`.
3. The OAuth credentials saved by the command-line tool can be readily used when making API calls using the package.

3.2 Twitter API

To ease configuration, birding adds a *`from_oauth_file()`* method which will creates a *Twitter* binding using the OAuth credential file created by the `twitter` command-line application. The `twitter` command need only be run once to create this file, which is saved in the user home directory at `~/.twitter_oauth`. Once that file is in place, twitter API interactions look like this:

- Twitter API Demo

3.3 Search Manager

It is useful to solve the problem itself before being concerned with details about the topology. birding's `TwitterSearchManager` composes the *Twitter* object into higher-level method signatures which perform the processing steps needed for the given *Problem statement & topology*. A full interaction before applying Storm looks like this (see `In[2]`):

- Simple Simulated Stream

3.4 Storm Bolts

With APIs in place to do the work, Bolt classes provide Storm components:

- *TwitterSearchBolt* searches the input terms.
- *TwitterLookupBolt* expands search results into full tweets.
- *ElasticsearchIndexBolt* indexes the lookup results in elasticsearch.
- *ResultTopicBolt* publishes the lookup results to Kafka.

3.5 Storm Spouts

Spout classes provide Storm components which take birding's input and provide the source of streams in the topology:

- *DispatchSpout()* dispatches spout class based on config. See *Configuring birding*.
- *TermCycleSpout* cycles through a static list of terms.

3.6 Storm Topology

With Storm components ready for streamparse, a topology can pull it all together. birding's topology uses the Clojure DSL; the streamparse discussion of topologies has more detail. In the topology definition below, note the class references `"birding.bolt.TwitterSearchBolt"`, `"birding.bolt.TwitterLookupBolt"`, and `"birding.bolt.ResultTopicBolt"`. These are full Python namespace references to the birding classes. The names given in the DSL can then be used to wire the components together. For example, the definition of `"search-bolt" (python-bolt-spec ...)` allows `"search-bolt"` to be used as input in another bolt, `"lookup-bolt" (python-bolt-spec ... {"search-bolt" :shuffle} ... )`.

```
(ns birding
  (:use     [streamparse.specs])
  (:gen-class))

(defn birding [options]
  [
    ;; spout configuration
    {"term-spout" (python-spout-spec
          options
          ; Dispatch class based on birding.yml.
          "birding.spout.DispatchSpout"
          ["term" "timestamp"]
          :conf {"topology.max.spout.pending", 8}
          )
    }
    ;; bolt configuration
    {"search-bolt" (python-bolt-spec
          options
          ; Use field grouping on term to support in-memory caching.
          {"term-spout" ["term"]}
          "birding.bolt.TwitterSearchBolt"
          ["term" "timestamp" "search_result"]
          :p 2
          )
     "lookup-bolt" (python-bolt-spec
          options
          {"search-bolt" :shuffle}
          "birding.bolt.TwitterLookupBolt"
          ["term" "timestamp" "lookup_result"]
          :p 2
```

```
          )
     "elasticsearch-index-bolt" (python-bolt-spec
          options
          {"lookup-bolt" :shuffle}
          "birding.bolt.ElasticsearchIndexBolt"
          []
          :p 1
          )
     "result-topic-bolt" (python-bolt-spec
          options
          {"lookup-bolt" :shuffle}
          "birding.bolt.ResultTopicBolt"
          []
          :p 1
          )
    }
  ]
)
```

Next, goto one of:

- *Downloading and running birding*
- *Using birding in production*
- *Configuring birding*

CHAPTER 4

Using *birding* in production

Note: birding is currently alpha software.

If birding itself satisfies project requirements, see the streamparse project's discussion of remote deployment and use `sparse submit` from a checkout of the birding repository. Otherwise, birding is available on the Python Package Index, which projects can use as a dependency:

```
pip install birding
```

Once installed in the Python environment, birding references are available to the topology definition. A project's topology can include `python-spout-spec` and `python-bolt-spec` declarations which have class references to `birding.spout` and `birding.bolt` namespaces, respectively. The snippet below illustrates this. The *Storm Topology* section has more detail.

```
"search-bolt" (python-bolt-spec
    options
    {"term-spout" ["term"]}
    "birding.bolt.TwitterSearchBolt"
    ["term" "timestamp" "search_result"]
    :p 2)
```

The streamparse project discusses remote deployment using the `sparse submit` command. *Configuring birding* discusses the `birding.yml` file which is located by the `BIRDING_CONF` environment variable. Projects using birding should include its configuration file as part of host configuration management or a streamparse submit hook, and likewise set the `BIRDING_CONF` variable accordingly.

Next, goto *Configuring birding*.

CHAPTER 5

Configuring *birding*

birding uses a validated configuration file for runtime details.

Configuration files use a YAML format. All values have a default (below) and accept values of the same name in the configuration file, which has a default path of `birding.yml` in the current working directory. If needed, the `BIRDING_CONF` environment variable can point to the filepath of the configuration file.

The scope of the configuration file is limited to details of birding itself, not of Storm-related topics. Storm details are in the project topology definition.

When a configuration value is a Python dotted name, it is a string reference to the Python object to import. In general, when the value is just an object name without a full namespace, its assumed to be the relevant birding namespace, e.g. `LRUShelf` is assumed to be `birding.shelf.LRUShelf`. Respective `*_init` configuration values specify keyword (not positional) arguments to be passed to the class constructor.

See *Using birding in production* for further discussion on configuration in production environments.

For advanced API usage, see *get_config()*. The config includes an *Appendix* to support any additional values not known to birding, such that these values are available in `config['Appendix']` and bypass any validation. This is useful for code which uses birding's config loader and needs to define additional values.

Defaults:

```
Spout: TermCycleSpout
TermCycleSpout:
  terms:
  - real-time analytics
  - apache storm
  - pypi
SearchManager:
  class: birding.twitter.TwitterSearchManagerFromOAuth
  init: {}
TwitterSearchBolt:
  shelf_class: FreshLRUShelf
  shelf_init: {}
  shelf_expiration: 300
ElasticsearchIndexBolt:
  elasticsearch_class: elasticsearch.Elasticsearch
  elasticsearch_init:
    hosts:
    - localhost: 9200
  index: tweet
  doc_type: tweet
ResultTopicBolt:
  kafka_class: pykafka.KafkaClient
```

```
  kafka_init:
    hosts: 127.0.0.1:9092 # comma-separated list of hosts
  topic: tweet
  shelf_class: ElasticsearchShelf
  shelf_init: {}
  shelf_expiration: null
Appendix: {}
```

CHAPTER 6

Searching Gnip

Gnip is Twitter's enterprise API platform, which birding supports for projects seeking to search at higher rates than allowed in the public API. The configuration snippet below uses Gnip's APIs instead of Twitter. See *Configuring birding* for how to configure birding.

```
SearchManager:
  class: birding.gnip.GnipSearchManager
  init:
    base_url: https://search.gnip.com/accounts/Example
    stream: prod.json
    username: admin@example.org
    password: This.yml.file.should.be.untracked.
```

See birding API docs for *Gnip* and *GnipSearchManager* for underlying behavior, which is minimal.

CHAPTER 7

API

birding.spout.**DispatchSpout**()
Factory to dispatch spout class based on config.

class birding.spout.**TermCycleSpout**

initialize(*stormconf*, *context*)
Initialization steps:

1.Prepare sequence of terms based on config: TermCycleSpout/terms.

next_tuple()
Next tuple steps:

1.Emit (term, timestamp) for next term in sequence w/current UTC time.

class birding.bolt.**TwitterSearchBolt**

initialize(*conf*, *ctx*)
Initialization steps:

1.Get *search_manager_from_config()*.

2.Prepare to track searched terms as to avoid redundant searches.

process(*tup*)
Process steps:

1.Stream in (term, timestamp).

2.Perform *search()* on term.

3.Emit (term, timestamp, search_result).

class birding.bolt.**TwitterLookupBolt**

initialize(*conf*, *ctx*)
Initialization steps:

1.Get *search_manager_from_config()*.

process(*tup*)
Process steps:

1.Stream in (term, timestamp, search_result).

2.Perform *lookup_search_result()*.

3.Emit (term, timestamp, lookup_result).

class birding.bolt.**ElasticsearchIndexBolt**

initialize(*conf*, *ctx*)
Initialization steps:

1.Prepare elasticsearch connection, including details for indexing.

process(*tup*)
Process steps:

1.Index third positional value from input to elasticsearch.

class birding.bolt.**ResultTopicBolt**

initialize(*conf*, *ctx*)
Initialization steps:

1.Connect to Kafka.

2.Prepare Kafka producer for *tweet* topic.

3.Prepare to track tweets published to topic, to avoid redundant data.

process(*tup*)
Process steps:

1.Stream third positional value from input into Kafka topic.

birding.search.**search_manager_from_config**()
Get a *SearchManager* instance dynamically based on config.

config is a dictionary containing `class` and `init` keys as defined in *birding.config*.

class birding.search.**SearchManager**
Abstract base class for service object to search for tweets.

lookup(*id_list*, ***kw*)
Lookup list of statuses, return results directly from source.

Input can be any sequence of numeric or string values representing status IDs.

lookup_search_result(*result*, ***kw*)
Perform *lookup()* on return value of *search()*.

search(*q=None*, ***kw*)
Search for `q`, return results directly from source.

class birding.twitter.**Twitter**(*format=u'json'*, *domain=u'api.twitter.com'*, *secure=True*, *auth=None*, *api_version=<class 'twitter.api._DEFAULT'>*, *retry=False*)

classmethod from_oauth_file(*filepath=None*)
Get an object bound to the Twitter API using your own credentials.

The *twitter* library ships with a *twitter* command that uses PIN OAuth. Generate your own OAuth credentials by running *twitter* from the shell, which will open a browser window to authenticate you. Once successfully run, even just one time, you will have a credential file at ~/.twitter_oauth.

This factory function reuses your credential file to get a *Twitter* object. (Really, this code is just lifted from the *twitter.cmdline* module to minimize OAuth dancing.)

class birding.twitter.**TwitterSearchManager**(*twitter*)
Service object to provide fully-hydrated tweets given a search query.

static dump(*result*)
Dump result into a string, useful for debugging.

lookup(*id_list*, ***kw*)
Lookup list of statuses, return results directly from twitter.

Input can be any sequence of numeric or string values representing twitter status IDs.

lookup_search_result(*result*, ***kw*)
Perform *lookup()* on return value of *search()*.

search(*q=None*, ***kw*)
Search twitter for q, return results directly from twitter.

birding.twitter.**TwitterSearchManagerFromOAuth**()
Build *TwitterSearchManager* from user OAuth file.

Arguments are passed to *birding.twitter.Twitter.from_oauth_file()*.

class birding.gnip.**Gnip**(*base_url*, *stream*, *username*, *password*, ***params*)
Simple binding to Gnip search API.

search(*q*, ***kw*)
Search Gnip for given query, returning deserialized response.

session_class
alias of Session

class birding.gnip.**GnipSearchManager**(**a*, ***kw*)
Service object to provide fully-hydrated tweets given a search query.

static dump(*result*)
Dump result into a string, useful for debugging.

lookup(*id_list*, ***kw*)
Not implemented.

lookup_search_result(*result*, ***kw*)
Do almost nothing, just pass-through results.

search(*q*, ***kw*)
Search gnip for q, return results directly from gnip.

birding.config.**get_config**(*filepath=None*, *default_loader=None*, *on_missing=None*)
Get a dict for the current birding configuration.

The resulting dictionary is fully populated with defaults, such that all valid keys will resolve to valid values. Invalid and extra values in the configuration result in an exception.

See *Configuring birding* (module-level docstring) for discussion on how birding configuration works, including filepath loading. Note that a non-default filepath set via env results in a OSError when the file is missing, but the default filepath is ignored when missing.

This function caches its return values as to only parse configuration once per set of inputs. As such, treat the resulting dictionary as read-only as not to accidentally write values which will be seen by other handles of the dictionary.

Parameters

- **filepath** (*str*) – path to birding configuration YAML file.

- **default_loader** (*callable*) – callable which returns file descriptor with YAML data of default configuration values
- **on_missing** (*callable*) – callback to call when file is missing.

Returns dict of current birding configuration; treat as read-only.

Return type dict

birding.shelf.**shelf_from_config**()
Get a *Shelf* instance dynamically based on config.

config is a dictionary containing `shelf_*` keys as defined in *birding.config*.

class birding.shelf.**Shelf**
Abstract base class for a shelf to track – but not iterate – values.

Provides a dict-interface.

clear()
Remove all items from the shelf.

delitem(*key*)
Remove an item from the shelf.

getitem(*key*)
Get an item's value from the shelf or raise KeyError(key).

pack(*key*, *value*)
Pack value given to `setitem`, inverse of `unpack`.

setitem(*key*, *value*)
Set an item on the shelf, with the given value.

unpack(*key*, *value*)
Unpack value from `getitem`.

This is useful for *Shelf* implementations which require metadata be stored with the shelved values, in which case `pack` should implement the inverse operation. By default, the value is simply passed through without modification. The `unpack` implementation is called on `__getitem__` and therefore can raise *KeyError* if packed metadata indicates that a value is invalid.

class birding.shelf.**FreshPacker**
Mixin for pack/unpack implementation to expire shelf content.

expire_after = 300
Values are no longer fresh after this value, in seconds.

freshness()
Clock function to use for freshness packing/unpacking.

is_fresh(*freshness*)
Return False if given freshness value has expired, else True.

pack(*key*, *value*)
Pack value with metadata on its freshness.

set_expiration(*expire_after*)
Set a new expiration for freshness of all unpacked values.

unpack(*key*, *value*)
Unpack and return value only if it is fresh.

class birding.shelf.**LRUShelf**(*maxsize=1000*)
An in-memory Least-Recently Used shelf up to *maxsize*..

class birding.shelf.**FreshLRUShelf**(*maxsize=1000*)
A Least-Recently Used shelf which expires values.

class birding.shelf.**ElasticsearchShelf**(*index='shelf'*, *doc_type='shelf'*, ***elasticsearch_init*)
A shelf implemented using an elasticsearch index.

class birding.shelf.**FreshElasticsearchShelf**(*index='shelf'*, *doc_type='shelf'*, ***elasticsearch_init*)
An shelf implementation with elasticsearch which expires values.

To discuss this project, join the streamparse user group.

Documentation Index

b

T

U