python-scrapinghub Documentation

Release 2.3.0

Pablo Hoffman, Daniel Graña

Dec 17, 2019

Contents

The `scrapinghub` is a Python library for communicating with the Scrapinghub API.

CHAPTER 1

Quickstart

1.1 Requirements

- Python 2.7 or above

1.2 Installation

The quick way:

```
pip install scrapinghub
```

It is recommended to install the library with MessagePack support, it provides better response time and improved bandwidth usage:

```
pip install scrapinghub[msgpack]
```

1.3 Basic usage

Instantiate a new client with your Scrapinghub API key:

```
>>> from scrapinghub import ScrapinghubClient
>>> apikey = '84c87545607a4bc0***************' # your API key as a string
>>> client = ScrapinghubClient(apikey)
```

Note: Your Scrapinghub API key is available at https://app.scrapinghub.com/account/apikey after you sign up with the service.

List your deployed projects:

```
>>> client.projects.list()
[123, 456]
```

Run a new job for one of your projects:

```
>>> project = client.get_project(123)
>>> project.jobs.run('spider1', job_args={'arg1': 'val1'})
<scrapinghub.client.Job at 0x106ee12e8>>
```

Access your job's output data:

```
>>> job = client.get_job('123/1/2')
>>> for item in job.items.iter():
...     print(item)
{
    'name': ['Some item'],
    'url': 'http://some-url/item.html',
    'value': 25,
}
{
    'name': ['Some other item'],
    'url': 'http://some-url/other-item.html',
    'value': 35,
}
...
```

Checkout all the other features in *Overview* or in the more detailed *API Reference*.

1.4 Tests

The package is covered with integration tests based on VCR.py library: there are recorded cassettes files in `tests/*/cassettes` used instead of HTTP requests to real services, it helps to simplify and speed up development.

By default, tests use VCR.py `once` mode to:

- replay previously recorded interactions.
- record new interactions if there is no cassette file.
- cause an error to be raised for new requests if there is a cassette file.

It means that if you add new integration tests and run all tests as usual, only new cassettes will be created, all existing cassettes will stay unmodified.

To ignore existing cassettes and use real services, please provide a flag:

```
py.test --ignore-cassettes
```

If you want to update/recreate all the cassettes from scratch, please use:

```
py.test --update-cassettes
```

Note that internally the above command erases the whole folder with cassettes.

CHAPTER 2

Overview

ScrapinghubClient is a Python client for communicating with the Scrapinghub API.

First, you instantiate a new client with your Scrapinghub API key:

```
>>> from scrapinghub import ScrapinghubClient
>>> apikey = '84c87545607a4bc0***************'
>>> client = ScrapinghubClient(apikey)
>>> client
<scrapinghub.client.ScrapinghubClient at 0x1047af2e8>
```

2.1 Working with projects

This client instance has a `projects` attribute for accessing your projects on Scrapinghub's platform.

With it, you can list the project IDs available in your account:

```
>>> client.projects.list()
[123, 456]
```

Note: `.list()` does not return *Project* instances, but their numeric IDs.

Or you can get a summary of all your projects (how many jobs are finished, running or pending to be run):

```
>>> client.projects.summary()
[{'finished': 674,
  'has_capacity': True,
  'pending': 0,
  'project': 123,
  'running': 1},
 {'finished': 33079,
  'has_capacity': True,
```

(continues on next page)

(continued from previous page)

```
  'pending': 0,
  'project': 456,
  'running': 2}]
```

To work with a particular project, reference it using its numeric ID:

```
>>> project = client.get_project(123)
>>> project
<scrapinghub.client.Project at 0x106cdd6a0>
>>> project.key
'123'
```

Note: `get_project()` returns a *Project* instance.

Tip: The above is a shortcut for `client.projects.get(123)`.

2.2 Working with spiders

A Scrapinghub project (usually) consists of a group of web crawlers called "spiders".

The different spiders within your project are accessible via the `spiders` attribute of the *Project* instance.

To get the list of spiders in the project, use `.spiders.list()`:

```
>>> project.spiders.list()
[
  {'id': 'spider1', 'tags': [], 'type': 'manual', 'version': '123'},
  {'id': 'spider2', 'tags': [], 'type': 'manual', 'version': '123'}
]
```

To select a particular spider to work with, use `.spiders.get(<spidername>)`:

```
>>> spider = project.spiders.get('spider2')
>>> spider
<scrapinghub.client.Spider at 0x106ee3748>
>>> spider.key
'123/2'
>>> spider.name
spider2
```

With `.spiders.get(<spidername>)`, you get a *Spider* instance back.

Note: `.spiders.list()` does not return *Spider* instances. The `id` key in the returned dicts corresponds to the `.name` attribute of *Spider* that you get with `.spiders.get(<spidername>)`.

2.3 Working with jobs collections

Essentially, the purpose of spiders is to be run in Scrapinghub's platform. Each spider run is called a "job". And a collection of spider jobs is represented by a *Jobs* object.

Both project-level jobs (i.e. all jobs from a project) and spider-level jobs (i.e. all jobs for a specific spider) are available as a `jobs` attribute of a *Project* instance or a *Spider* instance respectively.

2.3.1 Running jobs

Use the `.jobs.run()` method to run a new job for a project or a particular spider,:

```
>>> job = spider.jobs.run()
```

You can also use `.jobs.run()` at the project level, the difference being that a spider name is required:

```
>>> job = project.jobs.run('spider1')
```

Scheduling jobs supports different options, passed as arguments to `.run()`:

- **job_args** (dict): to provide arguments for the job
- **job_settings** (dict): to pass additional settings for the job
- **units** (integer): to specify amount of units to run the job
- **priority** (integer): to set higher/lower priority for the job
- **add_tag** (list of strings): to create a job with a set of initial tags
- **meta** (dict): to pass additional custom metadata

Check the run endpoint for more information.

For example, to run a new job for a given spider with custom parameters:

```
>>> job = spider.jobs.run(units=2, job_settings={'SETTING': 'VALUE'}, priority=1,
...                       add_tag=['tagA','tagB'], meta={'custom-data': 'val1'})
```

2.3.2 Geting job information

To select a specific job for a project, use `.jobs.get(<jobKey>)`:

```
>>> job = project.jobs.get('123/1/2')
>>> job.key
'123/1/2'
```

Also there's a shortcut to get same job with client instance:

```
>>> job = client.get_job('123/1/2')
```

These methods return a *Job* instance (see *below*).

2.3.3 Counting jobs

It's also possible to count jobs for a given project or spider via `.jobs.count()`:

```
>>> spider.jobs.count()
5
```

The counting logic supports different filters, as described for count endpoint.

2.3.4 Iterating over jobs

To loop over the spider jobs (most recently finished first), you can use `.jobs.iter()` to get an iterator object:

```
>>> jobs_summary = spider.jobs.iter()
>>> [j['key'] for j in jobs_summary]
['123/1/3', '123/1/2', '123/1/1']
```

The `.jobs.iter()` iterator generates dicts (not *Job* objects), e.g:

```
{u'close_reason': u'finished',
 u'elapsed': 201815620,
 u'finished_time': 1492843577852,
 u'items': 2,
 u'key': u'123320/3/155',
 u'logs': 21,
 u'pages': 2,
 u'pending_time': 1492843520319,
 u'running_time': 1492843526622,
 u'spider': u'spider001',
 u'state': u'finished',
 u'ts': 1492843563720,
 u'version': u'792458b-master'}
```

You typically use it like this:

```
>>> for job in jobs_summary:
...     # do something with job data
```

Or, if you just want to get the job IDs:

```
>>> [x['key'] for x in jobs_summary]
['123/1/3', '123/1/2', '123/1/1']
```

The job's dict fieldset from `.jobs.iter()` is less detailed than `job.metadata` (see below), but can contain a few additional fields as well, on demand. Additional fields can be requested using the `jobmeta` argument.

When `jobmeta` is used, the user MUST list all required fields, even default ones:

```
>>> # by default, the "spider" key is available in the dict from iter()
>>> job_summary = next(project.jobs.iter())
>>> job_summary.get('spider', 'missing')
'foo'
>>>
>>> # when jobmeta is use, if "spider" key is not listed in it,
>>> # iter() will not include "spider" key in the returned dicts
>>> jobs_summary = project.jobs.iter(jobmeta=['scheduled_by'])
```

(continues on next page)

(continued from previous page)

```
>>> job_summary = next(jobs_summary)
>>> job_summary.get('scheduled_by', 'missing')
'John'
>>> job_summary.get('spider', 'missing')
missing
```

By default `.jobs.iter()` returns the last 1000 jobs at most. To get more than the last 1000, you need to paginate through results in batches, using the `start` parameter:

```
>>> jobs_summary = spider.jobs.iter(start=1000)
```

There are several filters like `spider`, `state`, `has_tag`, `lacks_tag`, `startts` and `endts` (check list endpoint for more details).

To get jobs filtered by tags:

```
>>> jobs_summary = project.jobs.iter(has_tag=['new', 'verified'], lacks_tag='obsolete
↪')
```

Warning: The list of tags in `has_tag` is an *OR* condition, so in the case above, jobs with either `'new'` or `'verified'` tag are selected.

On the contrary the list of tags in `lacks_tag` is a logical *AND*.

To get a specific number of last finished jobs of some spider, use `spider`, `state` and `count` arguments:

```
>>> jobs_summary = project.jobs.iter(spider='foo', state='finished', count=3)
```

There are 4 possible job states, which can be used as (string) values for filtering by state:

- `'pending'`: the job is scheduled to run when enough units become available;
- `'running'`: the job is running;
- `'finished'`: the job has ended;
- `'deleted'`: the jobs has been deleted and will become unavailable when the platform performs its next cleanup.

Dictionary entries returned by `.jobs.iter()` method contain some additional meta, but can be easily converted to *Job* instances with:

```
>>> [Job(client, x['key']) for x in jobs]
[
  <scrapinghub.client.Job at 0x106e2cc18>,
  <scrapinghub.client.Job at 0x106e260b8>,
  <scrapinghub.client.Job at 0x106e26a20>,
]
```

2.3.5 Jobs summaries

To check jobs summary:

```
>>> spider.jobs.summary()
[{'count': 0, 'name': 'pending', 'summary': []},
 {'count': 0, 'name': 'running', 'summary': []},
 {'count': 5,
  'name': 'finished',
  'summary': [...]}
```

It's also possible to get last jobs summary (for each spider):

```
>>> list(sp.jobs.iter_last())
[{'close_reason': 'success',
  'elapsed': 3062444,
  'errors': 1,
  'finished_time': 1482911633089,
  'key': '123/1/3',
  'logs': 8,
  'pending_time': 1482911596566,
  'running_time': 1482911598909,
  'spider': 'spider1',
  'state': 'finished',
  'ts': 1482911615830,
  'version': 'some-version'}]
```

Note that there can be a lot of spiders, so the method above returns an iterator.

2.3.6 Updating tags

Tags is a convenient way to mark specific jobs (for better search, postprocessing etc).

To mark all spider jobs with tag `consumed`:

```
>>> spider.jobs.update_tags(add=['consumed'])
```

To remove existing tag `existing` for all spider jobs:

```
>>> spider.jobs.update_tags(remove=['existing'])
```

Modifying tags is available at *Spider* level and *Job* level.

2.3.7 Canceling jobs

To cancel a few jobs by keys at once:

```
>>> spider.jobs.cancel(['123/1/2', '123/1/3'])
```

All jobs should belong to the same project.

Note that there's a limit on amount of job keys you can cancel with a single call, please contact support if the amount is more than 1k.

2.4 Job actions

You can perform actions on a *Job* instance.

For example, to cancel a running or pending job, simply call `cancel()` on it:

```
>>> job.cancel()
```

To delete a job, its metadata, logs and items, call `delete()`:

```
>>> job.delete()
```

To mark a job with the tag `'consumed'`, call `update_tags()`:

```
>>> job.update_tags(add=['consumed'])
```

2.5 Job data

A *Job* instance provides access to its associated data, using the following attributes:

- `metadata`: various information on the job itself;
- `items`: the data items that the job produced;
- `logs`: log entries that the job produced;
- `requests`: HTTP requests that the job issued;
- `samples`: runtime stats that the job uploaded;

2.5.1 Metadata

Metadata about a job details can be accessed via its `metadata` attribute. The *corresponding object* acts like a Python dictionary:

```
>>> job.metadata.get('version')
'5123a86-master'
```

To check what keys are available (they ultimately depend on the job), you can use its `.iter()` method (here, it's wrapped inside a dict for readibility):

```
>>> dict(job.metadata.iter())
{...
 u'close_reason': u'finished',
 u'completed_by': u'jobrunner',
 u'deploy_id': 16,
 u'finished_time': 1493007370566,
 u'job_settings': {u'CLOSESPIDER_PAGECOUNT': 5,
                   u'SOME_CUSTOM_SETTING': 10},
 u'pending_time': 1493006433100,
 u'priority': 2,
 u'project': 123456,
 u'running_time': 1493006488829,
 u'scheduled_by': u'periodicjobs',
 u'scrapystats': {u'downloader/request_bytes': 96774,
                  u'downloader/request_count': 228,
                  u'downloader/request_method_count/GET': 228,
                  u'downloader/response_bytes': 923251,
                  u'downloader/response_count': 228,
                  u'downloader/response_status_count/200': 228,
```

(continues on next page)

(continued from previous page)

```
                  u'finish_reason': u'finished',
                  u'finish_time': 1493007337660.0,
                  u'httpcache/firsthand': 228,
                  u'httpcache/miss': 228,
                  u'httpcache/store': 228,
                  u'item_scraped_count': 684,
                  u'log_count/INFO': 22,
                  u'memusage/max': 63311872,
                  u'memusage/startup': 60248064,
                  u'request_depth_max': 50,
                  u'response_received_count': 228,
                  u'scheduler/dequeued': 228,
                  u'scheduler/dequeued/disk': 228,
                  u'scheduler/enqueued': 228,
                  u'scheduler/enqueued/disk': 228,
                  u'start_time': 1493006508701.0},
 u'spider': u'myspider',
 u'spider_args': {u'arg1': u'value1',
                  u'arg2': u'value2'},
 u'spider_type': u'manual',
 u'started_by': u'jobrunner',
 u'state': u'finished',
 u'tags': [],
 u'units': 1,
 u'version': u'792458b-master'}
```

As you may have noticed in the example above, if the job was a Scrapy spider run, the metadata object contains a special `'scrapystats'` key, which is a dict representation of the crawl's Scrapy stats values:

```
>>> job.metadata.get('scrapystats')
...
'downloader/response_count': 104,
'downloader/response_status_count/200': 104,
'finish_reason': 'finished',
'finish_time': 1447160494937,
'item_scraped_count': 50,
'log_count/DEBUG': 157,
'log_count/INFO': 1365,
'log_count/WARNING': 3,
'memusage/max': 182988800,
'memusage/startup': 62439424,
...
```

Anything can be stored in a job's metadata, here is example how to add tags:

```
>>> job.metadata.set('tags', ['obsolete'])
```

2.5.2 Items

To retrieve all scraped items (as Python dicts) from a job, use *job.items.iter()*:

```
>>> for item in job.items.iter():
...     # do something with item (it's just a dict)
```

2.5.3 Logs

To retrieve all log entries from a job use *job.logs.iter()*:

```
>>> for logitem in job.logs.iter():
...     # logitem is a dict with level, message, time
>>> logitem
{
  'level': 20,
  'message': '[scrapy.core.engine] Closing spider (finished)',
  'time': 1482233733976},
}
```

2.5.4 Requests

To retrieve all requests from a job, there's *job.requests.iter()*:

```
>>> for reqitem in job.requests.iter():
...     # reqitem is a dict
>>> reqitem
[{
  'duration': 354,
  'fp': '6d748741a927b10454c83ac285b002cd239964ea',
  'method': 'GET',
  'rs': 1270,
  'status': 200,
  'time': 1482233733870,
  'url': 'https://example.com'
}]
```

2.6 Project activity log

Project.activity provides a convenient interface to project activity events.

To retrieve activity events from a project, you can use `.activity.iter()`, with optional arguments (here, the last 3 events, with timestamp information):

```
>>> list(project.activity.iter(count=3, meta="_ts"))
[{u'_ts': 1493362000130,
  u'event': u'job:completed',
  u'job': u'123456/3/161',
  u'user': u'jobrunner'},
 {u'_ts': 1493361946077,
  u'event': u'job:started',
  u'job': u'123456/3/161',
  u'user': u'jobrunner'},
 {u'_ts': 1493361942440,
  u'event': u'job:scheduled',
  u'job': u'123456/3/161',
  u'user': u'periodicjobs'}]
```

To retrieve all the events, use `.activity.list()`

```
>>> project.activity.list()
[{'event': 'job:completed', 'job': '123/2/3', 'user': 'jobrunner'},
 {'event': 'job:cancelled', 'job': '123/2/3', 'user': 'john'}]
```

To post a new activity event, use `.activity.add()`:

```
>>> event = {'event': 'job:completed', 'job': '123/2/4', 'user': 'john'}
>>> project.activity.add(event)
```

Or post multiple events at once:

```
>>> events = [
...     {'event': 'job:completed', 'job': '123/2/5', 'user': 'john'},
...     {'event': 'job:cancelled', 'job': '123/2/6', 'user': 'john'},
... ]
>>> project.activity.add(events)
```

2.7 Collections

Scrapinghub's Collections provide a way to store an arbitrary number of records indexed by a key. They're often used by Scrapinghub projects as a single place to write information from multiple scraping jobs.

Read more about *Collections* in the official docs.

As an example, let's store a hash and timestamp pair for spider 'foo'.

The usual workflow with *`project.collections`* would be:

1. reference your project's `collections` attribute,
2. call `.get_store(<somename>)` to create or access the named collection you want (the collection will be created automatically if it doesn't exist) ; you get a "store" object back,
3. call `.set(<key/value> pairs)` to store data.

```
>>> collections = project.collections
>>> foo_store = collections.get_store('foo_store')
>>> foo_store.set({'_key': '002d050ee3ff6192dcbecc4e4b4457d7', 'value': '1447221694537
↪'})
>>> foo_store.count()
1
>>> foo_store.get('002d050ee3ff6192dcbecc4e4b4457d7')
{u'value': u'1447221694537'}
>>> # iterate over _key & value pair
... list(foo_store.iter())
[{u'_key': u'002d050ee3ff6192dcbecc4e4b4457d7', u'value': u'1447221694537'}]
>>> # filter by multiple keys - only values for keys that exist will be returned
... list(foo_store.iter(key=['002d050ee3ff6192dcbecc4e4b4457d7', 'blah']))
[{u'_key': u'002d050ee3ff6192dcbecc4e4b4457d7', u'value': u'1447221694537'}]
>>> foo_store.delete('002d050ee3ff6192dcbecc4e4b4457d7')
>>> foo_store.count()
0
```

Collections are available at project level only.

2.8 Frontiers

Typical workflow with *Frontiers*:

```
>>> frontiers = project.frontiers
```

Get all frontiers from a project to iterate through it:

```
>>> frontiers.iter()
<list_iterator at 0x103c93630>
```

List all frontiers:

```
>>> frontiers.list()
['test', 'test1', 'test2']
```

Get a *Frontier* instance by name:

```
>>> frontier = frontiers.get('test')
>>> frontier
<scrapinghub.client.Frontier at 0x1048ae4a8>
```

Get an iterator to iterate through a frontier slots:

```
>>> frontier.iter()
<list_iterator at 0x1030736d8>
```

List all slots:

```
>>> frontier.list()
['example.com', 'example.com2']
```

Get a *FrontierSlot* by name:

```
>>> slot = frontier.get('example.com')
>>> slot
<scrapinghub.client.FrontierSlot at 0x1049d8978>
```

Add a request to the slot:

```
>>> slot.queue.add([{'fp': '/some/path.html'}])
>>> slot.flush()
>>> slot.newcount
1
```

`newcount` is defined per slot, but also available per frontier and globally:

```
>>> frontier.newcount
1
>>> frontiers.newcount
3
```

Add a fingerprint only to the slot:

```
>>> slot.fingerprints.add(['fp1', 'fp2'])
>>> slot.flush()
```

There are convenient shortcuts: `f` for `fingerprints` to access *FrontierSlotFingerprints* and `q` for `queue` to access *FrontierSlotQueue*.

Add requests with additional parameters:

```
>>> slot.q.add([{'fp': '/'}, {'fp': 'page1.html', 'p': 1, 'qdata': {'depth': 1}}])
>>> slot.flush()
```

To retrieve all requests for a given slot:

```
>>> reqs = slot.q.iter()
```

To retrieve all fingerprints for a given slot:

```
>>> fps = slot.f.iter()
```

To list all the requests use `list()` method (similar for `fingerprints`):

```
>>> fps = slot.q.list()
```

To delete a batch of requests:

```
>>> slot.q.delete('00013967d8af7b0001')
```

To delete the whole slot from the frontier:

```
>>> slot.delete()
```

Flush data of the given frontier:

```
>>> frontier.flush()
```

Flush data of all frontiers of a project:

```
>>> frontiers.flush()
```

Close batch writers of all frontiers of a project:

```
>>> frontiers.close()
```

Frontiers are available on project level only.

2.9 Settings

You can work with project settings via *Settings*.

To get a list of the project settings:

```
>>> project.settings.list()
[(u'default_job_units', 2), (u'job_runtime_limit', 24)]]
```

To get a project setting value by name:

```
>>> project.settings.get('job_runtime_limit')
24
```

To update a project setting value by name:

```
>>> project.settings.set('job_runtime_limit', 20)
```

Or update a few project settings at once:

```
>>> project.settings.update({'default_job_units': 1,
...                          'job_runtime_limit': 20})
```

2.10 Exceptions

exception scrapinghub.**ScrapinghubAPIError**(*message=None*, *http_error=None*)
Base exception class.

exception scrapinghub.**BadRequest**(*message=None*, *http_error=None*)
Usually raised in case of 400 response from API.

exception scrapinghub.**Unauthorized**(*message=None*, *http_error=None*)
Request lacks valid authentication credentials for the target resource.

exception scrapinghub.**NotFound**(*message=None*, *http_error=None*)
Entity doesn't exist (e.g. spider or project).

exception scrapinghub.**ValueTooLarge**(*message=None*, *http_error=None*)
Value cannot be writtent because it exceeds size limits.

exception scrapinghub.**DuplicateJobError**(*message=None*, *http_error=None*)
Job for given spider with given arguments is already scheduled or running.

exception scrapinghub.**ServerError**(*message=None*, *http_error=None*)
Indicates some server error: something unexpected has happened.

CHAPTER 3

API Reference

3.1 Client object

class scrapinghub.client.**ScrapinghubClient**(*auth=None*, *dash_endpoint=None*, *connection_timeout=60*, ***kwargs*)

Main class to work with Scrapinghub API.

Parameters

- **auth** – (optional) Scrapinghub APIKEY or other SH auth credentials. If not provided, it will read, respectively, from SH_APIKEY or SHUB_JOBAUTH environment variables. SHUB_JOBAUTH is available by default in *Scrapy Cloud*, but it does not provide access to all endpoints (e.g. job scheduling), but it is allowed to access job data, collections, crawl frontier. If you need full access to *Scrapy Cloud* features, you'll need to provide a Scrapinghub APIKEY through this argument or deploying SH_APIKEY.
- **dash_endpoint** – (optional) Scrapinghub Dash panel url.
- ****kwargs** – (optional) Additional arguments for *HubstorageClient* constructor.

Variables **projects** – projects collection, *Projects* instance.

Usage:

```
>>> from scrapinghub import ScrapinghubClient
>>> client = ScrapinghubClient('APIKEY')
>>> client
<scrapinghub.client.ScrapinghubClient at 0x1047af2e8>
```

close(*timeout=None*)

Close client instance.

Parameters **timeout** – (optional) float timeout secs to stop gracefully.

get_job(*job_key*)

Get *Job* with a given job key.

Parameters **job_key** – job key string in format `project_id/spider_id/job_id`, where all the components are integers.

Returns a job instance.

Return type *Job*

Usage:

```
>>> job = client.get_job('123/1/1')
>>> job
<scrapinghub.client.jobs.Job at 0x10afe2eb1>
```

get_project(*project_id*)

Get *scrapinghub.client.projects.Project* instance with a given project id.

The method is a shortcut for client.projects.get().

Parameters **project_id** – integer or string numeric project id.

Returns a project instance.

Return type *Project*

Usage:

```
>>> project = client.get_project(123)
>>> project
<scrapinghub.client.projects.Project at 0x106cdd6a0>
```

3.2 Activity

class scrapinghub.client.activity.**Activity**(*cls*, *client*, *key*)

Representation of collection of job activity events.

Not a public constructor: use *Project* instance to get a *Activity* instance. See `activity` attribute.

Please note that *list()* method can use a lot of memory and for a large amount of activities it's recommended to iterate through it via *iter()* method (all params and available filters are same for both methods).

Usage:

- get all activity from a project:

```
>>> project.activity.iter()
<generator object jldecode at 0x1049ee990>
```

- get only last 2 events from a project:

```
>>> project.activity.list(count=2)
[{'event': 'job:completed', 'job': '123/2/3', 'user': 'jobrunner'},
 {'event': 'job:started', 'job': '123/2/3', 'user': 'john'}]
```

- post a new event:

```
>>> event = {'event': 'job:completed',
...          'job': '123/2/4',
...          'user': 'jobrunner'}
>>> project.activity.add(event)
```

- post multiple events at once:

```
>>> events = [
...     {'event': 'job:completed', 'job': '123/2/5', 'user': 'jobrunner'},
...     {'event': 'job:cancelled', 'job': '123/2/6', 'user': 'john'},
... ]
>>> project.activity.add(events)
```

add(*values*, ***kwargs*)
Add new event to the project activity.

Parameters **values** – a single event or a list of events, where event is represented with a dictionary of ('event', 'job', 'user') keys.

iter(*count=None*, ***params*)
Iterate over activity events.

Parameters **count** – limit amount of elements.

Returns a generator object over a list of activity event dicts.

Return type types.GeneratorType[dict]

list(**args*, ***kwargs*)
Convenient shortcut to list iter results.

Please note that *list()* method can use a lot of memory and for a large amount of elements it's recommended to iterate through it via *iter()* method (all params and available filters are same for both methods).

3.3 Collections

class scrapinghub.client.collections.**Collection**(*client*, *collections*, *type_*, *name*)
Representation of a project collection object.

Not a public constructor: use *Collections* instance to get a *Collection* instance. See *Collections.get_store()* and similar methods.

Usage:

- add a new item to collection:

```
>>> foo_store.set({'_key': '002d050ee3ff6192dcbecc4e4b4457d7',
...                'value': '1447221694537'})
```

- count items in collection:

```
>>> foo_store.count()
1
```

- get an item from collection:

```
>>> foo_store.get('002d050ee3ff6192dcbecc4e4b4457d7')
{'value': '1447221694537'}
```

- get all items from collection:

```
>>> foo_store.iter()
<generator object jldecode at 0x1049eef10>
```

- iterate over _key & value pair:

```
>>> for elem in foo_store.iter(count=1)):
...     print(elem)
[{'_key': '002d050ee3ff6192dcbecc4e4b4457d7', 'value': '1447221694537'}]
```

- get generator over item keys:

```
>>> keys = foo_store.iter(nodata=True, meta=["_key"]))
>>> next(keys)
{'_key': '002d050ee3ff6192dcbecc4e4b4457d7'}
```

- filter by multiple keys, only values for keys that exist will be returned:

```
>>> foo_store.list(key=['002d050ee3ff6192dcbecc4e4b4457d7', 'blah'])
[{'_key': '002d050ee3ff6192dcbecc4e4b4457d7', 'value': '1447221694537'}]
```

- delete an item by key:

```
>>> foo_store.delete('002d050ee3ff6192dcbecc4e4b4457d7')
```

- remove the entire collection with a single API call:

```
>>> foo_store.truncate()
```

count (**args*, ***kwargs*)

Count collection items with a given filters.

Returns amount of elements in collection.

Return type `int`

create_writer (*start=0*, *auth=None*, *size=1000*, *interval=15*, *qsize=None*, *content_encoding='identity'*, *maxitemsize=1048576*, *callback=None*)

Create a new writer for a collection.

Parameters

- **start** – (optional) initial offset for writer thread.
- **auth** – (optional) set auth credentials for the request.
- **size** – (optional) set initial queue size.
- **interval** – (optional) set interval for writer thread.
- **qsize** – (optional) setup max queue size for the writer.
- **content_encoding** – (optional) set different Content-Encoding header.
- **maxitemsize** – (optional) max item size in bytes.
- **callback** – (optional) some callback function.

Returns a new writer object.

Return type `scrapinghub.hubstorage.batchuploader._BatchWriter`

If provided - calllback shouldn't try to inject more items in the queue, otherwise it can lead to deadlocks.

delete (*keys*)

Delete item(s) from collection by key(s).

Parameters **keys** – a single key or a list of keys.

The method returns `None` (original method returns an empty generator).

get(*key*, ***params*)
Get item from collection by key.

Parameters

- **key** – string item key.
- ****params** – (optional) additional query params for the request.

Returns an item dictionary if exists.

Return type `dict`

iter(*key=None*, *prefix=None*, *prefixcount=None*, *startts=None*, *endts=None*, *requests_params=None*, ***params*)
A method to iterate through collection items.

Parameters

- **key** – a string key or a list of keys to filter with.
- **prefix** – a string prefix to filter items.
- **prefixcount** – maximum number of values to return per prefix.
- **startts** – UNIX timestamp at which to begin results.
- **endts** – UNIX timestamp at which to end results.
- **requests_params** – (optional) a dict with optional requests params.
- ****params** – (optional) additional query params for the request.

Returns an iterator over items list.

Return type `collections.Iterable[dict]`

list(*key=None*, *prefix=None*, *prefixcount=None*, *startts=None*, *endts=None*, *requests_params=None*, ***params*)
Convenient shortcut to list iter results.

Please note that *list()* method can use a lot of memory and for a large amount of logs it's recommended to iterate through it via *iter()* method (all params and available filters are same for both methods).

Parameters

- **key** – a string key or a list of keys to filter with.
- **prefix** – a string prefix to filter items.
- **prefixcount** – maximum number of values to return per prefix.
- **startts** – UNIX timestamp at which to begin results.
- **endts** – UNIX timestamp at which to end results.
- **requests_params** – (optional) a dict with optional requests params.
- ****params** – (optional) additional query params for the request.

Returns a list of items where each item is represented with a dict.

Return type `list[dict]`

set(*value*)
Set item to collection by key.

Parameters **value** – a dict representing a collection item.

The method returns `None` (original method returns an empty generator).

truncate()
Remove the entire collection with a single API call.

The method returns `None` (original method returns an empty generator).

class scrapinghub.client.collections.**Collections**(*cls*, *client*, *key*)
Access to project collections.

Not a public constructor: use *Project* instance to get a *Collections* instance. See `collections` attribute.

Usage:

```
>>> collections = project.collections
>>> collections.list()
[{'name': 'Pages', 'type': 's'}]
>>> foo_store = collections.get_store('foo_store')
```

get(*type_*, *name*)
Base method to get a collection with a given type and name.

Parameters

- **type_** – a collection type string.
- **name** – a collection name string.

Returns a collection object.

Return type *Collection*

get_cached_store(*name*)
Method to get a cashed-store collection by name.

The collection type means that items expire after a month.

Parameters **name** – a collection name string.

Returns a collection object.

Return type *Collection*

get_store(*name*)
Method to get a store collection by name.

Parameters **name** – a collection name string.

Returns a collection object.

Return type *Collection*

get_versioned_cached_store(*name*)
Method to get a versioned-cached-store collection by name.

Multiple copies are retained, and each one expires after a month.

Parameters **name** – a collection name string.

Returns a collection object.

Return type *Collection*

get_versioned_store(*name*)
Method to get a versioned-store collection by name.

The collection type retains up to 3 copies of each item.

Parameters **name** – a collection name string.

Returns a collection object.

Return type *Collection*

iter()
Iterate through collections of a project.

Returns an iterator over collections list where each collection is represented by a dictionary with ('name','type') fields.

Return type collections.Iterable[dict]

list()
List collections of a project.

Returns a list of collections where each collection is represented by a dictionary with ('name','type') fields.

Return type list[dict]

3.4 Exceptions

exception scrapinghub.client.exceptions.**BadRequest**(*message=None*, *http_error=None*)
Usually raised in case of 400 response from API.

exception scrapinghub.client.exceptions.**DuplicateJobError**(*message=None*, *http_error=None*)
Job for given spider with given arguments is already scheduled or running.

exception scrapinghub.client.exceptions.**Forbidden**(*message=None*, *http_error=None*)
You don't have the permission to access the requested resource. It is either read-protected or not readable by the server.

exception scrapinghub.client.exceptions.**NotFound**(*message=None*, *http_error=None*)
Entity doesn't exist (e.g. spider or project).

exception scrapinghub.client.exceptions.**ScrapinghubAPIError**(*message=None*, *http_error=None*)
Base exception class.

exception scrapinghub.client.exceptions.**ServerError**(*message=None*, *http_error=None*)
Indicates some server error: something unexpected has happened.

exception scrapinghub.client.exceptions.**Unauthorized**(*message=None*, *http_error=None*)
Request lacks valid authentication credentials for the target resource.

exception scrapinghub.client.exceptions.**ValueTooLarge**(*message=None*, *http_error=None*)
Value cannot be writtent because it exceeds size limits.

3.5 Frontiers

class scrapinghub.client.frontiers.**Frontier**(*client*, *frontiers*, *name*)

Representation of a frontier object.

Not a public constructor: use *Frontiers* instance to get a *Frontier* instance. See *Frontiers.get()* method.

Usage:

- get iterator with all slots:

```
>>> frontier.iter()
<list_iterator at 0x1030736d8>
```

- list all slots:

```
>>> frontier.list()
['example.com', 'example.com2']
```

- get a slot by name:

```
>>> frontier.get('example.com')
<scrapinghub.client.frontiers.FrontierSlot at 0x1049d8978>
```

- flush frontier data:

```
>>> frontier.flush()
```

- show amount of new requests added to frontier:

```
>>> frontier.newcount
3
```

flush()

Flush data for a whole frontier.

get(*slot*)

Get a slot by name.

Returns a frontier slot instance.

Return type *FrontierSlot*

iter()

Iterate through slots.

Returns an iterator over frontier slots names.

Return type collections.Iterable[str]

list()

List all slots.

Returns a list of frontier slots names.

Return type list[str]

newcount

Integer amount of new entries added to frontier.

class scrapinghub.client.frontiers.**FrontierSlot**(*client*, *frontier*, *slot*)

Representation of a frontier slot object.

Not a public constructor: use *Frontier* instance to get a *FrontierSlot* instance. See *Frontier.get()* method.

Usage:

- add request to a queue:

```
>>> data = [{'fp': 'page1.html', 'p': 1, 'qdata': {'depth': 1}}]
>>> slot.q.add('example.com', data)
```

- add fingerprints to a slot:

```
>>> slot.f.add(['fp1', 'fp2'])
```

- flush data for a slot:

```
>>> slot.flush()
```

- show amount of new requests added to a slot:

```
>>> slot.newcount
2
```

- read requests from a slot:

```
>>> slot.q.iter()
<generator object jldecode at 0x1049aa9e8>
>>> slot.q.list()
[{'id': '0115a8579633600006',
  'requests': [['page1.html', {'depth': 1}]]}]
```

- read fingerprints from a slot:

```
>>> slot.f.iter()
<generator object jldecode at 0x103de4938>
>>> slot.f.list()
['page1.html']
```

- delete a batch with requests from a slot:

```
>>> slot.q.delete('0115a8579633600006')
```

- delete a whole slot:

```
>>> slot.delete()
```

delete()

Delete the slot.

f

Shortcut to have quick access to slot fingerprints.

Returns fingerprints collection for the slot.

Return type *FrontierSlotFingerprints*

flush()

Flush data for the slot.

newcount
Integer amount of new entries added to slot.

q
Shortcut to have quick access to a slot queue.

Returns queue instance for the slot.

Return type *FrontierSlotQueue*

class scrapinghub.client.frontiers.**FrontierSlotFingerprints**(*slot*)
Representation of request fingerprints collection stored in slot.

add(*fps*)
Add new fingerprints to slot.

Parameters **fps** – a list of string fingerprints to add.

iter(***params*)
Iterate through fingerprints in the slot.

Parameters ****params** – (optional) additional query params for the request.

Returns an iterator over fingerprints.

Return type collections.Iterable[str]

list(***params*)
List fingerprints in the slot.

Parameters ****params** – (optional) additional query params for the request.

Returns a list of fingerprints.

Return type list[str]

class scrapinghub.client.frontiers.**FrontierSlotQueue**(*slot*)
Representation of request batches queue stored in slot.

add(*fps*)
Add requests to the queue.

delete(*ids*)
Delete request batches from the queue.

iter(*mincount=None*, ***params*)
Iterate through batches in the queue.

Parameters

- **mincount** – (optional) limit results with min amount of requests.
- ****params** – (optional) additional query params for the request.

Returns an iterator over request batches in the queue where each batch is represented with a dict with ('id', 'requests') field.

Return type collections.Iterable[dict]

list(*mincount=None*, ***params*)
List request batches in the queue.

Parameters

- **mincount** – (optional) limit results with min amount of requests.
- ****params** – (optional) additional query params for the request.

Returns a list of request batches in the queue where each batch is represented with a dict with ('id', 'requests') field.

Return type `list[dict]`

class `scrapinghub.client.frontiers.`**Frontiers**(**args*, ***kwargs*)

Frontiers collection for a project.

Not a public constructor: use *Project* instance to get a *Frontiers* instance. See `frontiers` attribute.

Usage:

- get all frontiers from a project:

```
>>> project.frontiers.iter()
<list_iterator at 0x103c93630>
```

- list all frontiers:

```
>>> project.frontiers.list()
['test', 'test1', 'test2']
```

- get a frontier by name:

```
>>> project.frontiers.get('test')
<scrapinghub.client.frontiers.Frontier at 0x1048ae4a8>
```

- flush data of all frontiers of a project:

```
>>> project.frontiers.flush()
```

- show amount of new requests added for all frontiers:

```
>>> project.frontiers.newcount
3
```

- close batch writers of all frontiers of a project:

```
>>> project.frontiers.close()
```

close()

Close frontier writer threads one-by-one.

flush()

Flush data in all frontiers writer threads.

get(*name*)

Get a frontier by name.

Parameters **name** – a frontier name string.

Returns a frontier instance.

Return type *Frontier*

iter()

Iterate through frontiers.

Returns an iterator over frontiers names.

Return type `collections.Iterable[str]`

list()
List frontiers names.

Returns a list of frontiers names.

Return type list[str]

newcount
Integer amount of new entries added to all frontiers.

3.6 Items

class scrapinghub.client.items.**Items**(*cls*, *client*, *key*)
Representation of collection of job items.

Not a public constructor: use *Job* instance to get a *Items* instance. See items attribute.

Please note that *list()* method can use a lot of memory and for a large number of items it's recommended to iterate through them via *iter()* method (all params and available filters are same for both methods).

Usage:

- retrieve all scraped items from a job:

```
>>> job.items.iter()
<generator object mpdecode at 0x10f5f3aa0>
```

- iterate through first 100 items and print them:

```
>>> for item in job.items.iter(count=100):
...     print(item)
```

- retrieve items with timestamp greater or equal to given timestamp (item here is an arbitrary dictionary depending on your code):

```
>>> job.items.list(startts=1447221694537)
[{
    'name': ['Some custom item'],
    'url': 'http://some-url/item.html',
    'size': 100000,
}]
```

- retrieve items via a generator of lists. This is most useful in cases where the job has a huge amount of items and it needs to be broken down into chunks when consumed. This example shows a job with 3 items:

```
>>> gen = job.items.list_iter(chunksize=2)
>>> next(gen)
[{'name': 'Item #1'}, {'name': 'Item #2'}]
>>> next(gen)
[{'name': 'Item #3'}]
>>> next(gen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- retrieving via meth::*list_iter* also supports the *start* and *count*. params. This is useful when you want to only retrieve a subset of items in a job. The example below belongs to a job with 10 items:

```
>>> gen = job.items.list_iter(chunksize=2, start=5, count=3)
>>> next(gen)
[{'name': 'Item #5'}, {'name': 'Item #6'}]
>>> next(gen)
[{'name': 'Item #7'}]
>>> next(gen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- retrieve 1 item with multiple filters:

```
>>> filters = [("size", ">", [30000]), ("size", "<", [40000])]
>>> job.items.list(count=1, filter=filters)
[{
    'name': ['Some other item'],
    'url': 'http://some-url/other-item.html',
    'size': 35000,
}]
```

close(*block=True*)
Close writers one-by-one.

flush()
Flush data from writer threads.

get(*key*, ***params*)
Get element from collection.

Parameters **key** – element key.

Returns a dictionary with element data.

Return type dict

iter(*_path=None*, *count=None*, *requests_params=None*, ***apiparams*)
A general method to iterate through elements.

Parameters **count** – limit amount of elements.

Returns an iterator over elements list.

Return type collections.Iterable

list(**args*, ***kwargs*)
Convenient shortcut to list iter results.

Please note that *list()* method can use a lot of memory and for a large amount of elements it's recommended to iterate through it via *iter()* method (all params and available filters are same for both methods).

list_iter(*chunksize=1000*, **args*, ***kwargs*)
An alternative interface for reading items by returning them as a generator which yields lists of items sized as *chunksize*.

This is a convenient method for cases when processing a large amount of items from a job isn't ideal in one go due to the large memory needed. Instead, this allows you to process it chunk by chunk.

You can improve I/O overheads by increasing the chunk value but that would also increase the memory consumption.

Parameters

- **chunksize** – size of list to be returned per iteration
- **start** – offset to specify the start of the item iteration
- **count** – overall number of items to be returned, which is broken down by *chunksize*.

Returns an iterator over items, yielding lists of items.

Return type `collections.Iterable`

stats()
Get resource stats.

Returns a dictionary with stats data.

Return type `dict`

write(*item*)
Write new element to collection.

Parameters **item** – element data dict to write.

3.7 Jobs

class `scrapinghub.client.jobs.`**Job**(*client*, *job_key*)
Class representing a job object.

Not a public constructor: use *ScrapinghubClient* instance or *Jobs* instance to get a *Job* instance. See *scrapinghub.client.ScrapinghubClient.get_job()* and *Jobs.get()* methods.

Variables

- **project_id** – integer project id.
- **key** – a job key.
- **items** – *Items* resource object.
- **logs** – *Logs* resource object.
- **requests** – *Requests* resource object.
- **samples** – *Samples* resource object.
- **metadata** – *JobMeta* resource object.

Usage:

```
>>> job = project.jobs.get('123/1/2')
>>> job.key
'123/1/2'
>>> job.metadata.get('state')
'finished'
```

cancel()
Schedule a running job for cancellation.

Usage:

```
>>> job.cancel()
>>> job.metadata.get('cancelled_by')
'John'
```

close_writers()
Stop job batch writers threads gracefully.

Called on `ScrapinghubClient.close()` method.

delete(***params*)
Mark finished job for deletion.

Parameters ****params** – (optional) keyword meta parameters to update.

Returns a previous string job state.

Return type `str`

Usage:

```
>>> job.delete()
'finished'
```

finish(***params*)
Move running job to finished state.

Parameters ****params** – (optional) keyword meta parameters to update.

Returns a previous string job state.

Return type `str`

Usage:

```
>>> job.finish()
'running'
```

start(***params*)
Move job to running state.

Parameters ****params** – (optional) keyword meta parameters to update.

Returns a previous string job state.

Return type `str`

Usage:

```
>>> job.start()
'pending'
```

update(*state*, ***params*)
Update job state.

Parameters

- **state** – a new job state.
- ****params** – (optional) keyword meta parameters to update.

Returns a previous string job state.

Return type `str`

Usage:

```
>>> job.update('finished')
'running'
```

update_tags(*add=None*, *remove=None*)
Partially update job tags.

It provides a convenient way to mark specific jobs (for better search, postprocessing etc).

Parameters

- **add** – (optional) list of tags to add.
- **remove** – (optional) list of tags to remove.

Usage: to mark a job with tag `consumed`:

```
>>> job.update_tags(add=['consumed'])
```

class scrapinghub.client.jobs.**JobMeta**(*cls*, *client*, *key*)
Class representing job metadata.

Not a public constructor: use *Job* instance to get a *JobMeta* instance. See `metadata` attribute.

Usage:

- get job metadata instance:

```
>>> job.metadata
<scrapinghub.client.jobs.JobMeta at 0x10494f198>
```

- iterate through job metadata:

```
>>> job.metadata.iter()
<dict_itemiterator at 0x104adbd18>
```

- list job metadata:

```
>>> job.metadata.list()
[('project', 123), ('units', 1), ('state', 'finished'), ...]
```

- get meta field value by name:

```
>>> job.metadata.get('version')
'test'
```

- update job meta field value (some meta fields are read-only):

```
>>> job.metadata.set('my-meta', 'test')
```

- update multiple meta fields at once

```
>>> job.metadata.update({'my-meta1': 'test1', 'my-meta2': 'test2'})
```

- delete meta field by name:

```
>>> job.metadata.delete('my-meta')
```

delete(*key*)
Delete element by key.

Parameters **key** – a string key

get(*key*)
Get element value by key.

Parameters **key** – a string key

iter()
Iterate through key/value pairs.

Returns an iterator over key/value pairs.

Return type collections.Iterable

list(**args*, ***kwargs*)
Convenient shortcut to list iter results.

Please note that *list()* method can use a lot of memory and for a large amount of elements it's recommended to iterate through it via *iter()* method (all params and available filters are same for both methods).

set(*key*, *value*)
Set element value.

Parameters

- **key** – a string key
- **value** – new value to set for the key

update(*values*)
Update multiple elements at once.

The method provides convenient interface for partial updates.

Parameters **values** – a dictionary with key/values to update.

class scrapinghub.client.jobs.**Jobs**(*client*, *project_id*, *spider=None*)
Class representing a collection of jobs for a project/spider.

Not a public constructor: use *Project* instance or *Spider* instance to get a *Jobs* instance. See scrapinghub.client.projects.Project.jobs and scrapinghub.client.spiders.Spider.jobs attributes.

Variables

- **project_id** – a string project id.
- **spider** – *Spider* object if defined.

Usage:

```
>>> project.jobs
<scrapinghub.client.jobs.Jobs at 0x10477f0b8>
>>> spider = project.spiders.get('spider1')
>>> spider.jobs
<scrapinghub.client.jobs.Jobs at 0x104767e80>
```

cancel(*keys=None*, *count=None*, ***params*)
Cancel a list of jobs using the keys provided.

Parameters

- **keys** – (optional) a list of strings containing the job keys in the format: <project>/<spider>/<job_id>.
- **count** – (optional) it requires admin access. Used for admins to bulk cancel an amount of count jobs.

Returns a dict with the amount of jobs cancelled.

Return type `dict`

Usage:

- cancel jobs 123 and 321 from project 111 and spiders 222 and 333:

```
>>> project.jobs.cancel(['111/222/123', '111/333/321'])
{'count': 2}
```

- cancel 100 jobs asynchronously:

```
>>> project.jobs.cancel(count=100)
{'count': 100}
```

count(*spider=None*, *state=None*, *has_tag=None*, *lacks_tag=None*, *startts=None*, *endts=None*, ***params*)

Count jobs with a given set of filters.

Parameters

- **spider** – (optional) filter by spider name.
- **state** – (optional) a job state, a string or a list of strings.
- **has_tag** – (optional) filter results by existing tag(s), a string or a list of strings.
- **lacks_tag** – (optional) filter results by missing tag(s), a string or a list of strings.
- **startts** – (optional) UNIX timestamp at which to begin results, in millisecons.
- **endts** – (optional) UNIX timestamp at which to end results, in millisecons.
- ****params** – (optional) other filter params.

Returns jobs count.

Return type `int`

The endpoint used by the method counts only finished jobs by default, use `state` parameter to count jobs in other states.

Usage:

```
>>> spider = project.spiders.get('spider1')
>>> spider.jobs.count()
5
>>> project.jobs.count(spider='spider2', state='finished')
2
```

get(*job_key*)

Get a *Job* with a given job_key.

Parameters **job_key** – a string job key.

job_key's project component should match the project used to get *Jobs* instance, and job_key's spider component should match the spider (if *Spider* was used to get *Jobs* instance).

Returns a job object.

Return type *Job*

Usage:

```
>>> job = project.jobs.get('123/1/2')
>>> job.key
'123/1/2'
```

iter(*count=None*, *start=None*, *spider=None*, *state=None*, *has_tag=None*, *lacks_tag=None*, *startts=None*, *endts=None*, *meta=None*, ***params*)

Iterate over jobs collection for a given set of params.

Parameters

- **count** – (optional) limit amount of returned jobs.
- **start** – (optional) number of jobs to skip in the beginning.
- **spider** – (optional) filter by spider name.
- **state** – (optional) a job state, a string or a list of strings.
- **has_tag** – (optional) filter results by existing tag(s), a string or a list of strings.
- **lacks_tag** – (optional) filter results by missing tag(s), a string or a list of strings.
- **startts** – (optional) UNIX timestamp at which to begin results, in millisecons.
- **endts** – (optional) UNIX timestamp at which to end results, in millisecons.
- **meta** – (optional) request for additional fields, a single field name or a list of field names to return.
- ****params** – (optional) other filter params.

Returns a generator object over a list of dictionaries of jobs summary for a given filter params.

Return type `types.GeneratorType[dict]`

The endpoint used by the method returns only finished jobs by default, use `state` parameter to return jobs in other states.

Usage:

- retrieve all jobs for a spider:

```
>>> spider.jobs.iter()
<generator object jldecode at 0x1049bd570>
```

- get all job keys for a spider:

```
>>> jobs_summary = spider.jobs.iter()
>>> [job['key'] for job in jobs_summary]
['123/1/3', '123/1/2', '123/1/1']
```

- job summary fieldset is less detailed than *JobMeta* but contains a few new fields as well. Additional fields can be requested using `meta` parameter. If it's used, then it's up to the user to list all the required fields, so only few default fields would be added except requested ones:

```
>>> jobs_summary = project.jobs.iter(meta=['scheduled_by', ])
```

- by default *Jobs.iter()* returns maximum last 1000 results. Pagination is available using start parameter:

```
>>> jobs_summary = spider.jobs.iter(start=1000)
```

- get jobs filtered by tags (list of tags has `OR` power):

```
>>> jobs_summary = project.jobs.iter(
...     has_tag=['new', 'verified'], lacks_tag='obsolete')
```

- get certain number of last finished jobs per some spider:

```
>>> jobs_summary = project.jobs.iter(
...     spider='spider2', state='finished', count=3)
```

iter_last(*start=None*, *start_after=None*, *count=None*, *spider=None*, ***params*)
Iterate through last jobs for each spider.

Parameters

- **start** – (optional)
- **start_after** – (optional)
- **count** – (optional)
- **spider** – (optional) a spider name (not needed if instantiated with *Spider*).
- ****params** – (optional) additional keyword args.

Returns a generator object over a list of dictionaries of jobs summary for a given filter params.

Return type types.GeneratorType[dict]

Usage:

- get all last job summaries for a project:

```
>>> project.jobs.iter_last()
<generator object jldecode at 0x1048a95c8>
```

- get last job summary for a a spider:

```
>>> list(spider.jobs.iter_last())
[{'close_reason': 'success',
  'elapsed': 3062444,
  'errors': 1,
  'finished_time': 1482911633089,
  'key': '123/1/3',
  'logs': 8,
  'pending_time': 1482911596566,
  'running_time': 1482911598909,
  'spider': 'spider1',
  'state': 'finished',
  'ts': 1482911615830,
  'version': 'some-version'}]
```

list(*count=None*, *start=None*, *spider=None*, *state=None*, *has_tag=None*, *lacks_tag=None*, *startts=None*, *endts=None*, *meta=None*, ***params*)
Convenient shortcut to list iter results.

Parameters

- **count** – (optional) limit amount of returned jobs.
- **start** – (optional) number of jobs to skip in the beginning.
- **spider** – (optional) filter by spider name.
- **state** – (optional) a job state, a string or a list of strings.

- **has_tag** – (optional) filter results by existing tag(s), a string or a list of strings.
- **lacks_tag** – (optional) filter results by missing tag(s), a string or a list of strings.
- **startts** – (optional) UNIX timestamp at which to begin results, in millisecons.
- **endts** – (optional) UNIX timestamp at which to end results, in millisecons.
- **meta** – (optional) request for additional fields, a single field name or a list of field names to return.
- ****params** – (optional) other filter params.

Returns list of dictionaries of jobs summary for a given filter params.

Return type `list[dict]`

The endpoint used by the method returns only finished jobs by default, use `state` parameter to return jobs in other states.

Please note that *`list()`* can use a lot of memory and for a large amount of logs it's recommended to iterate through it via *`iter()`* method (all params and available filters are same for both methods).

run(*spider=None*, *units=None*, *priority=None*, *meta=None*, *add_tag=None*, *job_args=None*, *job_settings=None*, *cmd_args=None*, *environment=None*, ***params*)
Schedule a new job and returns its job key.

Parameters

- **spider** – a spider name string (not needed if job is scheduled via `Spider.jobs`).
- **units** – (optional) amount of units for the job.
- **priority** – (optional) integer priority value.
- **meta** – (optional) a dictionary with metadata.
- **add_tag** – (optional) a string tag or a list of tags to add.
- **job_args** – (optional) a dictionary with job arguments.
- **job_settings** – (optional) a dictionary with job settings.
- **cmd_args** – (optional) a string with script command args.
- **environment** – (option) a dictionary with custom environment
- ****params** – (optional) additional keyword args.

Returns a job instance, representing the scheduled job.

Return type *`Job`*

Usage:

```
>>> job = project.jobs.run('spider1', job_args={'arg1': 'val1'})
>>> job
<scrapinghub.client.jobs.Job at 0x7fcb7c01df60>
>>> job.key
'123/1/1'
```

summary(*state=None*, *spider=None*, ***params*)
Get jobs summary (optionally by state).

Parameters

- **state** – (optional) a string state to filter jobs.

- **spider** – (optional) a spider name (not needed if instantiated with *Spider*).
- ****params** – (optional) additional keyword args.

Returns a list of dictionaries of jobs summary for a given filter params grouped by job state.

Return type `list[dict]`

Usage:

```
>>> spider.jobs.summary()
[{'count': 0, 'name': 'pending', 'summary': []},
 {'count': 0, 'name': 'running', 'summary': []},
 {'count': 5, 'name': 'finished', 'summary': [...]}

>>> project.jobs.summary('pending')
{'count': 0, 'name': 'pending', 'summary': []}
```

update_tags(*add=None*, *remove=None*, *spider=None*)

Update tags for all existing spider jobs.

Parameters

- **add** – (optional) list of tags to add to selected jobs.
- **remove** – (optional) list of tags to remove from selected jobs.
- **spider** – (optional) spider name, must if used with `Project.jobs`.

It's not allowed to update tags for all project jobs, so spider must be specified (it's done implicitly when using `Spider.jobs`, or you have to specify `spider` param when using `Project.jobs`).

Returns amount of jobs that were updated.

Return type `int`

Usage:

- mark all spider jobs with tag `consumed`:

```
>>> spider = project.spiders.get('spider1')
>>> spider.jobs.update_tags(add=['consumed'])
5
```

- remove existing tag `existing` for all spider jobs:

```
>>> project.jobs.update_tags(
...     remove=['existing'], spider='spider2')
2
```

3.8 Logs

class scrapinghub.client.logs.**Logs**(*cls*, *client*, *key*)

Representation of collection of job logs.

Not a public constructor: use *Job* instance to get a *Logs* instance. See `logs` attribute.

Please note that *list()* method can use a lot of memory and for a large amount of logs it's recommended to iterate through it via *iter()* method (all params and available filters are same for both methods).

Usage:

- retrieve all logs from a job:

```
>>> job.logs.iter()
<generator object mpdecode at 0x10f5f3aa0>
```

- iterate through first 100 log entries and print them:

```
>>> for log in job.logs.iter(count=100):
...     print(log)
```

- retrieve a single log entry from a job:

```
>>> job.logs.list(count=1)
[{
    'level': 20,
    'message': '[scrapy.core.engine] Closing spider (finished)',
    'time': 1482233733976,
}]
```

- retrive logs with a given log level and filter by a word:

```
>>> filters = [("message", "contains", ["mymessage"])]
>>> job.logs.list(level='WARNING', filter=filters)
[{
    'level': 30,
    'message': 'Some warning: mymessage',
    'time': 1486375511188,
}]
```

batch_write_start()
Override to set a start parameter when commencing writing.

close(*block=True*)
Close writers one-by-one.

debug(*message*, ***other*)
Log a message with DEBUG level.

error(*message*, ***other*)
Log a message with ERROR level.

flush()
Flush data from writer threads.

get(*key*, ***params*)
Get element from collection.

Parameters **key** – element key.

Returns a dictionary with element data.

Return type dict

info(*message*, ***other*)
Log a message with INFO level.

iter(*_path=None*, *count=None*, *requests_params=None*, ***apiparams*)
A general method to iterate through elements.

Parameters **count** – limit amount of elements.

Returns an iterator over elements list.

Return type collections.Iterable

list(**args*, ***kwargs*)
Convenient shortcut to list iter results.

Please note that *list()* method can use a lot of memory and for a large amount of elements it's recommended to iterate through it via *iter()* method (all params and available filters are same for both methods).

log(*message*, *level=20*, *ts=None*, ***other*)
Base method to write a log entry.

Parameters

- **message** – a string message.
- **level** – (optional) logging level, default to INFO.
- **ts** – (optional) UNIX timestamp in milliseconds.
- ****other** – other optional kwargs.

stats()
Get resource stats.

Returns a dictionary with stats data.

Return type dict

warn(*message*, ***other*)
Log a message with WARN level.

warning(*message*, ***other*)
Log a message with WARN level.

write(*item*)
Write new element to collection.

Parameters **item** – element data dict to write.

3.9 Projects

class scrapinghub.client.projects.**Project**(*client*, *project_id*)
Class representing a project object and its resources.

Not a public constructor: use *ScrapinghubClient* instance or *Projects* instance to get a *Project* instance. See *scrapinghub.client.ScrapinghubClient.get_project()* or *Projects.get()* methods.

Variables

- **key** – string project id.
- **activity** – *Activity* resource object.
- **collections** – *Collections* resource object.
- **frontiers** – *Frontiers* resource object.
- **jobs** – *Jobs* resource object.
- **settings** – Settings resource object.
- **spiders** – *Spiders* resource object.

Usage:

```
>>> project = client.get_project(123)
>>> project
<scrapinghub.client.projects.Project at 0x106cdd6a0>
>>> project.key
'123'
```

class scrapinghub.client.projects.**Projects**(*client*)

Collection of projects available to current user.

Not a public constructor: use *ScrapinghubClient* client instance to get a *Projects* instance. See scrapinghub.client.Scrapinghub.projects attribute.

Usage:

```
>>> client.projects
<scrapinghub.client.projects.Projects at 0x1047ada58>
```

get(*project_id*)

Get project for a given project id.

Parameters **project_id** – integer or string numeric project id.

Returns a project object.

Return type *Project*

Usage:

```
>>> project = client.projects.get(123)
>>> project
<scrapinghub.client.projects.Project at 0x106cdd6a0>
```

iter()

Iterate through list of projects available to current user.

Provided for the sake of API consistency.

Returns an iterator over project ids list.

Return type collections.Iterable[int]

list()

Get list of projects available to current user.

Returns a list of project ids.

Return type list[int]

Usage:

```
>>> client.projects.list()
[123, 456]
```

summary(*state=None*, ***params*)

Get short summaries for all available user projects.

Parameters **state** – a string state or a list of states.

Returns a list of dictionaries: each dictionary represents a project summary (amount of pending/running/finished jobs and a flag if it has a capacity to run new jobs).

Return type list[dict]

Usage:

```
>>> client.projects.summary()
[{'finished': 674,
  'has_capacity': True,
  'pending': 0,
  'project': 123,
  'running': 1},
 {'finished': 33079,
  'has_capacity': True,
  'pending': 0,
  'project': 456,
  'running': 2}]
```

class scrapinghub.client.projects.**Settings**(*cls*, *client*, *key*)

Class representing job metadata.

Not a public constructor: use *Project* instance to get a *Settings* instance. See `Project.settings` attribute.

Usage:

- get project settings instance:

```
>>> project.settings
<scrapinghub.client.projects.Settings at 0x10ecf1250>
```

- iterate through project settings:

```
>>> project.settings.iter()
<dictionary-itemiterator at 0x10ed11578>
```

- list project settings:

```
>>> project.settings.list()
[(u'default_job_units', 2), (u'job_runtime_limit', 20)]
```

- get setting value by name:

```
>>> project.settings.get('default_job_units')
2
```

- update setting value (some settings are read-only):

```
>>> project.settings.set('default_job_units', 2)
```

- update multiple settings at once:

```
>>> project.settings.update({'default_job_units': 1,
...                          'job_runtime_limit': 20})
```

- delete project setting by name:

```
>>> project.settings.delete('job_runtime_limit')
```

delete(*key*)

Delete element by key.

Parameters **key** – a string key

get(*key*)
Get element value by key.

Parameters **key** – a string key

iter()
Iterate through key/value pairs.

Returns an iterator over key/value pairs.

Return type collections.Iterable

list(**args*, ***kwargs*)
Convenient shortcut to list iter results.

Please note that *list()* method can use a lot of memory and for a large amount of elements it's recommended to iterate through it via *iter()* method (all params and available filters are same for both methods).

set(*key*, *value*)
Update project setting value by key.

Parameters

- **key** – a string setting key.
- **value** – new setting value.

update(*values*)
Update multiple elements at once.

The method provides convenient interface for partial updates.

Parameters **values** – a dictionary with key/values to update.

3.10 Requests

class scrapinghub.client.requests.**Requests**(*cls*, *client*, *key*)
Representation of collection of job requests.

Not a public constructor: use *Job* instance to get a *Requests* instance. See requests attribute.

Please note that *list()* method can use a lot of memory and for a large amount of logs it's recommended to iterate through it via *iter()* method (all params and available filters are same for both methods).

Usage:

- retrieve all requests from a job:

```
>>> job.requests.iter()
<generator object mpdecode at 0x10f5f3aa0>
```

- iterate through the requests:

```
>>> for reqitem in job.requests.iter(count=1):
...     print(reqitem['time'])
1482233733870
```

- retrieve single request from a job:

```
>>> job.requests.list(count=1)
[{
    'duration': 354,
    'fp': '6d748741a927b10454c83ac285b002cd239964ea',
    'method': 'GET',
    'rs': 1270,
    'status': 200,a
    'time': 1482233733870,
    'url': 'https://example.com'
}]
```

add(*url*, *status*, *method*, *rs*, *duration*, *ts*, *parent=None*, *fp=None*)
Add a new requests.

Parameters

- **url** – string url for the request.
- **status** – HTTP status of the request.
- **method** – stringified request method.
- **rs** – response body length.
- **duration** – request duration in milliseconds.
- **ts** – UNIX timestamp in milliseconds.
- **parent** – (optional) parent request id.
- **fp** – (optional) string fingerprint for the request.

close(*block=True*)
Close writers one-by-one.

flush()
Flush data from writer threads.

get(*key*, ***params*)
Get element from collection.

Parameters **key** – element key.

Returns a dictionary with element data.

Return type `dict`

iter(*_path=None*, *count=None*, *requests_params=None*, ***apiparams*)
A general method to iterate through elements.

Parameters **count** – limit amount of elements.

Returns an iterator over elements list.

Return type `collections.Iterable`

list(**args*, ***kwargs*)
Convenient shortcut to list iter results.

Please note that *list()* method can use a lot of memory and for a large amount of elements it's recommended to iterate through it via *iter()* method (all params and available filters are same for both methods).

stats()
Get resource stats.

Returns a dictionary with stats data.

Return type `dict`

write(*item*)
Write new element to collection.

Parameters **item** – element data dict to write.

3.11 Samples

class scrapinghub.client.samples.**Samples**(*cls*, *client*, *key*)
Representation of collection of job samples.

Not a public constructor: use *Job* instance to get a *Samples* instance. See `samples` attribute.

Please note that *list()* method can use a lot of memory and for a large amount of logs it's recommended to iterate through it via *iter()* method (all params and available filters are same for both methods).

Usage:

- retrieve all samples from a job:

```
>>> job.samples.iter()
<generator object mpdecode at 0x10f5f3aa0>
```

- retrieve samples with timestamp greater or equal to given timestamp:

```
>>> job.samples.list(startts=1484570043851)
[[1484570043851, 554, 576, 1777, 821, 0],
 [1484570046673, 561, 583, 1782, 821, 0]]
```

close(*block=True*)
Close writers one-by-one.

flush()
Flush data from writer threads.

get(*key*, ***params*)
Get element from collection.

Parameters **key** – element key.

Returns a dictionary with element data.

Return type `dict`

iter(*_key=None*, *count=None*, ***params*)
Iterate over elements in collection.

Parameters **count** – limit amount of elements.

Returns a generator object over a list of element dictionaries.

Return type `types.GeneratorType[dict]`

list(**args*, ***kwargs*)
Convenient shortcut to list iter results.

Please note that *list()* method can use a lot of memory and for a large amount of elements it's recommended to iterate through it via *iter()* method (all params and available filters are same for both methods).

stats()
Get resource stats.

Returns a dictionary with stats data.

Return type dict

write(*item*)
Write new element to collection.

Parameters **item** – element data dict to write.

3.12 Spiders

class scrapinghub.client.spiders.**Spider**(*client*, *project_id*, *spider_id*, *spider*)
Class representing a Spider object.

Not a public constructor: use *Spiders* instance to get a *Spider* instance. See *Spiders.get()* method.

Variables

- **project_id** – a string project id.
- **key** – a string key in format 'project_id/spider_id'.
- **name** – a spider name string.
- **jobs** – a collection of jobs, *Jobs* object.

Usage:

```
>>> spider = project.spiders.get('spider1')
>>> spider.key
'123/1'
>>> spider.name
'spider1'
```

list_tags(***kwargs*)
List spider tags.

Returns a list of spider tags.

Return type list[str]

update_tags(***kwargs*)
Update tags for the spider.

Parameters

- **add** – (optional) a list of string tags to add.
- **remove** – (optional) a list of string tags to remove.

class scrapinghub.client.spiders.**Spiders**(*client*, *project_id*)
Class to work with a collection of project spiders.

Not a public constructor: use *Project* instance to get a *Spiders* instance. See spiders attribute.

Variables **project_id** – string project id.

Usage:

```
>>> project.spiders
<scrapinghub.client.spiders.Spiders at 0x1049ca630>
```

get(*spider*, ***params*)

Get a spider object for a given spider name.

The method gets/sets spider id (and checks if spider exists).

Parameters **spider** – a string spider name.

Returns a spider object.

Return type *scrapinghub.client.spiders.Spider*

Usage:

```
>>> project.spiders.get('spider2')
<scrapinghub.client.spiders.Spider at 0x106ee3748>
>>> project.spiders.get('non-existing')
NotFound: Spider non-existing doesn't exist.
```

iter()

Iterate through a list of spiders for a project.

Returns an iterator over spiders list where each spider is represented as a dict containing its metadata.

Return type collection.Iterable[dict]

Provided for the sake of API consistency.

list()

Get a list of spiders for a project.

Returns a list of dictionaries with spiders metadata.

Return type list[dict]

Usage:

```
>>> project.spiders.list()
[{'id': 'spider1', 'tags': [], 'type': 'manual', 'version': '123'},
 {'id': 'spider2', 'tags': [], 'type': 'manual', 'version': '123'}]
```

CHAPTER 4

Legacy clients

4.1 scrapinghub.Connection

The module is the very first Python library for communicating with the Scrapinghub API.

[WARNING] It is deprecated, please use scrapinghub.ScrapinghubClient instead.

4.1.1 Overview

First, you connect to Scrapinghub:

```
>>> from scrapinghub import Connection
>>> conn = Connection('APIKEY')
>>> conn
Connection('APIKEY')
```

You can list the projects available to your account:

```
>>> conn.project_ids()
[123, 456]
```

And select a particular project to work with:

```
>>> project = conn[123]
>>> project
Project(Connection('APIKEY'), 123)
>>> project.id
123
```

To schedule a spider run (it returns the job id):

```
>>> project.schedule('myspider', arg1='val1')
u'123/1/1'
```

To get the list of spiders in the project:

```
>>> project.spiders()
[
  {u'id': u'spider1', u'tags': [], u'type': u'manual', u'version': u'123'},
  {u'id': u'spider2', u'tags': [], u'type': u'manual', u'version': u'123'}
]
```

To get all finished jobs:

```
>>> jobs = project.jobs(state='finished')
```

`jobs` is a `JobSet`. `JobSet` objects are iterable and, when iterated, return an iterable of `Job` objects, so you typically use it like this:

```
>>> for job in jobs:
...     # do something with job
```

Or, if you just want to get the job ids:

```
>>> [x.id for x in jobs]
[u'123/1/1', u'123/1/2', u'123/1/3']
```

To select a specific job:

```
>>> job = project.job(u'123/1/2')
>>> job.id
u'123/1/2'
```

To retrieve all scraped items from a job:

```
>>> for item in job.items():
...     # do something with item (it's just a dict)
```

To retrieve all log entries from a job:

```
>>> for logitem in job.log():
...     # logitem is a dict with logLevel, message, time
```

To get job info:

```
>>> job.info['spider']
'myspider'
>>> job.info['started_time']
'2010-09-28T15:09:57.629000'
>>> job.info['tags']
[]
>>> job.info['fields_count]['description']
1253
```

To mark a job with tag `consumed`:

```
>>> job.update(add_tag='consumed')
```

To mark several jobs with tag `consumed` (`JobSet` also supports the `update()` method):

```
>>> project.jobs(state='finished').update(add_tag='consumed')
```

To delete a job:

```
>>> job.delete()
```

To delete several jobs (`JobSet` also supports the `update()` method):

```
>>> project.jobs(state='finished').delete()
```

4.1.2 Module contents

Scrapinghub API Client Library

exception scrapinghub.legacy.**APIError**(*message*, *_type=None*)

Bases: exceptions.Exception

ERR_AUTH_ERROR = 'err_auth_error'

ERR_BAD_REQUEST = 'err_bad_request'

ERR_DEFAULT = 'err_default'

ERR_NOT_FOUND = 'err_not_found'

ERR_SERVER_ERROR = 'err_server_error'

ERR_VALUE_ERROR = 'err_value_error'

class scrapinghub.legacy.**Connection**(*apikey=None*, *password=''*, *_old_passwd=''*, *url=None*, *connection_timeout=None*)

Bases: object

Main class to access Scrapinghub API.

API_METHODS = {'addversion': 'scrapyd/addversion', 'as_project_slybot': 'as/project-

DEFAULT_ENDPOINT = 'https://app.scrapinghub.com/api/'

auth

project_ids()

Returns a list of projects available for this connection and crendentials.

project_names()

class scrapinghub.legacy.**Job**(*project*, *id*, *info*)

Bases: *scrapinghub.legacy.RequestProxyMixin*

MAX_RETRIES = 180

RETRY_INTERVAL = 60

add_report(*key*, *content*, *content_type='text/plain'*)

delete()

id

items(*offset=0*, *count=None*, *meta=None*)

log(***params*)

stop()

update(***modifiers*)

class scrapinghub.legacy.**JobSet**(*project*, ***params*)
Bases: *scrapinghub.legacy.RequestProxyMixin*

count()
Returns total results count of current filters. Does not inclue *count* neither *offset*.

delete()

stop()

update(***modifiers*)

class scrapinghub.legacy.**Project**(*connection*, *projectid*)
Bases: *scrapinghub.legacy.RequestProxyMixin*

autoscraping_project_slybot(*spiders=()*, *outputfile=None*)

autoscraping_spider_properties(*spider*, *start_urls=None*)

job(*id*)

jobs(***params*)

name

schedule(*spider*, ***params*)

spiders(***params*)

class scrapinghub.legacy.**RequestProxyMixin**
Bases: object

4.2 scrapinghub.HubstorageClient

The library can be used for interaction with spiders, jobs and scraped data through `storage.scrapinghub.com` endpoints.

[WARNING] It is deprecated, please use scrapinghub.ScrapinghubClient instead.

4.2.1 Overview

First, use your API key for authorization:

```
>>> from scrapinghub import HubstorageClient
>>> hc = HubstorageClient(auth='apikey')
>>> hc.server_timestamp()
1446222762611
```

Project

To get project settings or jobs summary:

```
>>> project = hc.get_project('1111111')
>>> project.settings['botgroups']
[u'botgroup1', ]
>>> project.jobsummary()
{u'finished': 6,
 u'has_capacity': True,
```

(continues on next page)

(continued from previous page)

```
 u'pending': 0,
 u'project': 1111111,
 u'running': 0}
```

Spider

To get spider id correlated with its name:

```
>>> project.ids.spider('foo')
1
```

To see last jobs summaries:

```
>>> summaries = project.spiders.lastjobsummary(count=3)
```

To get job summary per spider:

```
>>> summary = project.spiders.lastjobsummary(spiderid='1')
```

Job

Job can be **retrieved** directly by id (project_id/spider_id/job_id):

```
>>> job = hc.get_job('1111111/1/1')
>>> job.key
'1111111/1/1'
>>> job.metadata['state']
u'finished'
```

Creating a new job requires a spider name:

```
>>> job = hc.push_job(projectid='1111111', spidername='foo')
>>> job.key
'1111111/1/1'
```

Priority can be between 0 and 4 (from lowest to highest), the default is 2.

To push job from project level with the highest priority:

```
>>> job = project.push_job(spidername='foo', priority=4)
>>> job.metadata['priority']
4
```

Pushing a job with spider arguments:

```
>>> project.push_job(spidername='foo', spider_args={'arg1': 'foo', 'arg2': 'bar'})
```

Running job can be **cancelled** by calling `request_cancel()`:

```
>>> job.request_cancel()
>>> job.metadata['cancelled_by']
u'John'
```

To **delete** job:

```
>>> job.purged()
>>> job.metadata['state']
u'deleted'
```

Job details

Job details can be found in jobs metadata and it's scrapystats:

```
>>> job = hc.get_job('1111111/1/1')
>>> job.metadata['version']
u'5123a86-master'
>>> job.metadata['scrapystats']
...
u'downloader/response_count': 104,
u'downloader/response_status_count/200': 104,
u'finish_reason': u'finished',
u'finish_time': 1447160494937,
u'item_scraped_count': 50,
u'log_count/DEBUG': 157,
u'log_count/INFO': 1365,
u'log_count/WARNING': 3,
u'memusage/max': 182988800,
u'memusage/startup': 62439424,
...
```

Anything can be stored in metadata, here is example how to add tags:

```
>>> job.update_metadata({'tags': 'obsolete'})
```

Jobs

To iterate through all jobs metadata per project (descending order):

```
>>> jobs_metadata = project.jobq.list()
>>> [j['key'] for j in jobs_metadata]
['1111111/1/3', '1111111/1/2', '1111111/1/1']
```

Jobq metadata fieldset is less detailed, than `job.metadata`, but contains few new fields as well. Additional fields can be requested using the `jobmeta` parameter. If it used, then it's up to the user to list all the required fields, so only few default fields would be added except requested ones:

```
>>> metadata = next(project.jobq.list())
>>> metadata.get('spider', 'missing')
u'foo'
>>> jobs_metadata = project.jobq.list(jobmeta=['scheduled_by'])
>>> metadata = next(jobs_metadata)
>>> metadata.get('scheduled_by', 'missing')
u'John'
>>> metadata.get('spider', 'missing')
missing
```

By default `jobq.list()` returns maximum last 1000 results. Pagination is available using the `start` parameter:

```
>>> jobs_metadata = project.jobq.list(start=1000)
```

There are several filters like spider, state, has_tag, lacks_tag, startts and endts. To get jobs filtered by tags:

```
>>> jobs_metadata = project.jobq.list(has_tag=['new', 'verified'], lacks_tag='obsolete
↪')
```

List of tags has `OR` power, so in the case above jobs with 'new' or 'verified' tag are expected.

To get certain number of last finished jobs per some spider:

```
>>> jobs_metadata = project.jobq.list(spider='foo', state='finished', count=3)
```

There are 4 possible job states, which can be used as values for filtering by state:

- pending
- running
- finished
- deleted

Items

To iterate through items:

```
>>> items = job.items.iter_values()
>>> for item in items:
...     # do something, item is just a dict
```

Logs

To iterate through 10 first logs for example:

```
>>> logs = job.logs.iter_values(count=10)
>>> for log in logs:
...     # do something, log is a dict with log level, message and time keys
```

Collections

Let's store hash and timestamp pair for foo spider. Usual workflow with *Collections* would be:

```
>>> collections = project.collections
>>> foo_store = collections.new_store('foo_store')
>>> foo_store.set({'_key': '002d050ee3ff6192dcbecc4e4b4457d7', 'value': '1447221694537
↪'})
>>> foo_store.count()
1
>>> foo_store.get('002d050ee3ff6192dcbecc4e4b4457d7')
{u'value': u'1447221694537'}
>>> # iterate over _key & value pair
... list(foo_store.iter_values())
[{u'_key': u'002d050ee3ff6192dcbecc4e4b4457d7', u'value': u'1447221694537'}]
>>> # filter by multiple keys - only values for keys that exist will be returned
... list(foo_store.iter_values(key=['002d050ee3ff6192dcbecc4e4b4457d7', 'blah']))
[{u'_key': u'002d050ee3ff6192dcbecc4e4b4457d7', u'value': u'1447221694537'}]
```

(continues on next page)

(continued from previous page)

```
>>> foo_store.delete('002d050ee3ff6192dcbecc4e4b4457d7')
>>> foo_store.count()
0
```

Frontier

Typical workflow with *Frontier*:

```
>>> frontier = project.frontier
```

Add a request to the frontier:

```
>>> frontier.add('test', 'example.com', [{'fp': '/some/path.html'}])
>>> frontier.flush()
>>> frontier.newcount
1
```

Add requests with additional parameters:

```
>>> frontier.add('test', 'example.com', [{'fp': '/'}, {'fp': 'page1.html', 'p': 1,
↪'qdata': {'depth': 1}}])
>>> frontier.flush()
>>> frontier.newcount
2
```

To delete the slot `example.com` from the frontier:

```
>>> frontier.delete_slot('test', 'example.com')
```

To retrieve requests for a given slot:

```
>>> reqs = frontier.read('test', 'example.com')
```

To delete a batch of requests:

```
>>> frontier.delete('test', 'example.com', '00013967d8af7b0001')
```

To retrieve fingerprints for a given slot:

```
>>> fps = [req['requests'] for req in frontier.read('test', 'example.com')]
```

4.2.2 Module contents

HubStorage client library

class scrapinghub.hubstorage.**HubstorageClient**(*auth=None*, *endpoint=None*, *connection_timeout=None*, *max_retries=None*, *max_retry_time=None*, *user_agent=None*, *use_msgpack=True*)

Bases: object

DEFAULT_CONNECTION_TIMEOUT_S = 60.0

DEFAULT_ENDPOINT = 'https://storage.scrapinghub.com/'

DEFAULT_USER_AGENT = 'python-scrapinghub/2.3.0'

RETRY_DEFAULT_EXPONENTIAL_BACKOFF_MS = 500

RETRY_DEFAULT_JITTER_MS = 500

RETRY_DEFAULT_MAX_RETRIES = 3

RETRY_DEFAUT_MAX_RETRY_TIME_S = 60.0

batchuploader

close(*timeout=None*)

get_job(**args*, ***kwargs*)

get_project(**args*, ***kwargs*)

push_job(*projectid*, *spidername*, *auth=None*, ***jobparams*)

request(*is_idempotent=False*, ***kwargs*)

Execute an HTTP request with the current client session.

Use the retry policy configured in the client when is_idempotent is True

server_timestamp()

CHAPTER 5

Release notes

5.1 2.3.0 (2019-12-17)

- add *items.list_iter* method to iterate by chunks
- fix retrying logic for HTTP errors
- improve documentation

5.2 2.2.1 (2019-08-07)

- use *jobs.cancel* command name to maintain consistency
- provide basic documentation for the new feature

5.3 2.2.0 (2019-08-06)

- add a command to cancel multiple jobs per call
- normalize and simplify using VCR.py cassettes in tests

5.4 2.1.1 (2019-04-25)

- add Python 3.7 support
- update msgpack dependency
- fix *iter* logic for items/requests/logs
- add *truncate* method to collections

- improve documentation

5.5 2.1.0 (2019-01-14)

- add an option to schedule jobs with custom environment variables
- fallback to *SHUB_JOBAUTH* environment variable if *SH_APIKEY* is not set
- provide a unified connection timeout used by both internal clients
- increase a chunk size when working with the items stats endpoint

Python 3.3 is considered unmaintained.

5.6 2.0.3 (2017-12-08)

- fix *iter* logic when applying single *count* param

5.7 2.0.2 (2017-12-05)

- add support for TZ-aware datetime objects
- better tests

5.8 2.0.1 (2017-07-19)

- add a client parameter to disable msgpack use
- add VCR.py json-serialized tests
- make *parent* param optional for requests.add
- improve documentation

5.9 2.0.0 (2017-03-29)

Major release with a lot of new features.

- new powerfull ScrapinghubClient takes best from Connection and HubstorageClient, and combines it under single interface
- documentation is available on Read The Docs (2.0.0)

5.10 1.9.0 (2016-11-02)

- python-hubstorage merged into python-scrapinghub
- all tests are improved and rewritten with py.test
- hubstorage tests use vcrpy cassettes, work faster and don't require any external services to run

python-hubstorage is going to be considered deprecated, its next version will contain a deprecation warning and a proposal to use python-scrapinghub >=1.9.0 instead.

5.11 1.8.0 (2016-07-29)

- python 3 support & unittests
- add retries on httplib.HTTPException
- update scrapinghub api endpoint

5.12 1.7.0 (2014-07-25)

- basic py3.3 compatibility while keeping py2.7 compatibility
- update setup.py classifiers

5.13 1.6.2 (2014-07-01)

- fix travis workaround deploying on tags

5.14 1.6.1 (2014-07-01)

- packaging improvements
- cleaner implementation of project.job()

5.15 1.6.0 (2014-03-14)

- support retreiving a fixed amount of items

5.16 1.5.0 (2014-01-29)

- switch to dash secure endpoint

5.17 1.4.4 (2013-12-18)

- log download failure as error only if all attempts exhausted

5.18 1.4.3 (2013-11-25)

- update travis config to match travis-ci (pypy updated to 2.2)
- update pypi credentials

5.19 1.4.2 (2013-11-25)

- add python 3 to travis-ci matrix

5.20 1.4.1 (2013-11-25)

- tox, travis-ci and pypi uploads
- pypi uploads only on Python 2.7 success
- run tests under pypy 2.1 in travis-ci

5.21 1.4.0 (2013-09-04)

- add bindings for autoscraping api

5.22 1.3.0 (2013-08-26)

- add a way to set starting offset
- suport requesting meta fields

5.23 1.2.1 (2013-08-22)

- resume item downloads on network errors

5.24 1.2.0 (2013-08-08)

- add support for stopping a job
- project.name is deprecated in favour of project.id
- use stricter arguments for Connection constructor
- point to dash.scrapinghub.com api endpoint by default
- enable streaming with requests >= 1.0

5.25 1.1.1 (2012-10-24)

- added automatic retry to items download, when the request fails

5.26 1.1 (2012-10-19)

- report correct version on user-agent string
- ported to uses Requests library (instead of urllib2)
- added support for gzip transfer encoding to increase API throughput on low bandwidth connections
- deprecated first url argument of scrapinghub.Connection object
- added support for loading API key from SH_APIKEY environment variable

5.27 0.1 (2011-08-15)

First release of python-scrapinghub.

Python Module Index

s

Index

E

F

G

H

I

J

L

M

N

P

Q

R

S

T

U

V

W