Scrapyd API Documentation

Release 2.1.2

Darian Moody

April 01, 2018

Contents

`python-scrapyd-api` is a very simple Python wrapper for working with Scrapyd's API; it allows a Python application to talk to, and therefore control, the Scrapy Daemon.

It is built on top of the Requests library and supports Python 2.6, 2.7, 3.3 & 3.4.

CHAPTER 1

Contents

Installation

The package is available via the Python Package Index and can be installed in the usual ways:

```
$ easy_install python-scrapyd-api
```

or:

```
$ pip install python-scrapyd-api
```

Usage Instructions

Quick Usage

Please see the README for quick usage instructions.

Instantiating the wrapper

The wrapper is the core component which allows you to talk to Scrapyd's API and in most cases this will be the first and only point of interaction with this package.

```
from scrapyd_api import ScrapydAPI
scrapyd = ScrapydAPI('http://localhost:6800')
```

Where `http://localhost:6800` is the absolute URI to the location of the service, including which port Scrapyd's API is running on.

Note that while it is usually better to be explicit, if you are running Scrapyd on the same machine and with the default port then the wrapper can be instantiated with no arguments at all.

You may have further special requirements, for example one of the following:

- you may require HTTP Basic Authentication for your connections to Scrapyd.
- you may have changed the default endpoints for the various API actions.
- you may need to swap out the default connection client/handler.
- you may want to provide a timeout for client requests so the program does not hang indefinitely in case the server is not responding.

Providing HTTP Basic Auth credentials

The `auth` parameter can be passed during instantiation. The value itself should be a tuple containing two strings: the username and password required to successfully authenticate:

```
credentials = ('admin-username', 'admin-p4ssw0rd')
scrapyd = ScrapydAPI('http//example.com:6800/scrapyd/', auth=credentials)
```

Note: If you pass the `client` argument explained below, the `auth` argument's value will be ignored. The `auth` argument is only meant as a shortcut for setting the credentials on the built-in client; therefore, when passing your own you will have to handle this yourself.

Supplying custom endpoints for the various API actions

You might have changed the location of the *add version* action for whatever reason, here is how you would go about overriding that so the wrapper contacts the correct endpoint:

```
from scrapyd_api.constants import ADD_VERSION_ENDPOINT

custom_endpoints = {
    ADD_VERSION_ENDPOINT: '/changed-add-version-location.json'
}
scrapyd = ScrapydAPI('http://localhost:6800', endpoints=custom_endpoints)
```

The code example above only overrides the *add version endpoint* location and thus for all other endpoints, the default value would still be utilised. Simply add extra endpoint keys to the dict you pass in to override extra endpoints; the key values can either:

- be imported from `scrapyd_api.constants` as per the example.
- or simply set as strings, see the constants module for correct usage.

Replacing the client used

When no `client` argument is passed, the wrapper uses the default client which can be found at `scrapyd_api.client.Client`. This default client is effectively a small modification of Requests' `Session` client which knows how to handle errors from Scrapyd in a more graceful fashion by raising a `ScrapydResponseError` exception.

If you have custom authentication requirements or other issues which the default client does not solve then you can create your own client class, instantiate it and then pass it in to the constructor. This may be as simple as subclassing `scrapyd_api.client.Client` and modifying its functionality or it may require building your own Request's Session-like class. It would be done like so:

```
new_client = SomeNewClient()
scrapyd = ScrapydAPI('http://localhost:6800', client=new_client)
```

At the very minimum the client object should support:

- the `.get()` and `.post()` methods which should accept Requests-list args.
- the responses being parsed in a similar fashion to the `scrapd_api.client.Client._handle_response` method which has the ability to load the JSON returned and check the "status" which gets sent from Scrapyd, raising the `ScrapydResponseError` exception as required.

Setting timeout for the requests

By default, client requests do not time out unless a timeout value is set explicitly. Thus, if the server is not responding, your code may hang indefinitely. You can tell the client to stop waiting for a response after a given number of seconds with the `timeout` parameter provided during instantiation of the wrapper:

```
scrapyd = ScrapydAPI('http//example.com:6800/scrapyd/', timeout=5)
```

The value should be a float or a (connect timeout, read timeout) tuple. It will be supplied to every request to the server. Additional information can be found in the Requests documentation.

Calling the API

The Scrapyd API has a number of different actions designed to enable the full control and automation of the daemon itself, and this package provides a wrapper for *all* of those.

Add a version

`ScrapydAPI.`**`add_version`**(*project*, *version*, *egg*)

Uploads a new version of a project. See the add version endpoint on Scrapyd's documentation.

Arguments:

- **project** *(string)* The name of the project.
- **version** *(string)* The name of the new version you are uploading.
- **egg** *(string)* The Python egg you wish to upload as the project, as a pre-opened file.

Returns: *(int)* The number of spiders found in the uploaded project; this is the only useful information returned by Scrapyd as part of this call.

```
>>> with open('some-egg.egg') as egg:
>>>     scrapyd.add_version('project_name', 'version_name', egg)
3
```

Cancel a job

`ScrapydAPI.`**`cancel`**(*project*, *job*, *signal=None*)

Cancels a running or pending job with an optionally supplied termination signal. A job in this regard is a previously scheduled run of a specific spider. See the cancel endpoint on Scrapyd's documentation.

Arguments:

- **project** *(string)* The name of the project the job belongs to.
- **job** *(string)* The ID of the job (which was reported back on scheduling).
- **signal** *(optional - string or int)* The termination signal to use. If one is not provided, this field is not send allowing scrapyd to pick the default.

Returns: *(string)* `'running'` if the cancelled job was active, or `'pending'` if it was waiting to run.

```
>>> scrapyd.cancel('project_name', 'a3cb2..4efc1')
'running'
>>> scrapyd.cancel('project_name', 'b3ea2..3acc2', signal='TERM')
'pending'
```

Delete a project

`ScrapydAPI.`**`delete_project`**(*project*)

Deletes all versions of an entire project, this includes all spiders within those versions. See the delete project endpoint on Scrapyd's documentation.

Arguments:

- **project** *(string)* The name of the project to delete.

Returns: *(bool)* Always True, an exception is raised for other outcomes.

```
>>> scrapyd.delete_project('project_name')
True
```

Delete a version of a project

`ScrapydAPI.`**`delete_version`**(*project*, *version*)

Deletes a specific version of a project and all spiders within that version. See the delete version endpoint on Scrapyd's documentation.

Arguments:

- **project** *(string)* The name of the project which the version belongs to.
- **version** *(string)* The name of the version you wish to delete.

Returns: *(bool)* Always True, an exception is raised for other outcomes.

```
>>> scrapyd.delete_version('project_name', 'version_name')
True
```

Retrieve the status of a specific job

`ScrapydAPI.`**`job_status`**(*project*, *job_id*)

New in version 0.2.

Returns the job status for a single job. The status returned can be one of: `''`, `'running'`, `'pending'` or `'finished'`. The empty string is returned if the job ID could not be found and the status is therefore unknown.

Arguments:

- **project** *(string)* The name of the project which the version belongs to.
- **job_id** *(string)* The ID of the job you wish to check the status of.

Returns: *(string)* The status of the job, if known.

Note: Scrapyd does not support an endpoint for this specific action. This method's result is derived from the list jobs endpoint, and therefore this is a helper method/shortcut provided by this wrapper itself. This is why the call requires the *project* argument, as the list jobs endpoint underlying this method also requires it.

```
>>> scrapyd.job_status('project_name', 'ac32a..bc21')
'running'
```

If you wish, the various strings defining job state can be imported from the `scrapyd` module itself for use in comparisons. e.g:

```
from scrapyd_api import RUNNING, FINISHED, PENDING

state = scrapyd.job_status('project_name', 'ac32a..bc21')
if state == RUNNING:
    print 'Job is running'
```

List all jobs for a project

`ScrapydAPI.`**`list_jobs`**`(`*`project`*`)`

Lists all running, finished & pending spider jobs for a given project. See the list jobs endpoint on Scrapyd's documentation.

- **project** *(string)* The name of the project to list jobs for.

Returns: *(dict)* A dictionary with keys `pending`, `running` and `finished`, each containing a list of job dicts. Each job dict has keys for the `id` and the name of the `spider` which ran the job.

```
>>> scrapyd.list_jobs('project_name')
{
    'pending': [
        {
            u'id': u'24c35...f12ae',
            u'spider': u'spider_name'
        },
    ],
    'running': [
        {
            u'id': u'14a65...b27ce',
            u'spider': u'spider_name',
            u'start_time': u'2014-06-17 22:45:31.975358'
        },
    ],
    'finished': [
        {
            u'id': u'34c23...b21ba',
            u'spider': u'spider_name',
            u'start_time': u'2014-06-17 22:45:31.975358',
            u'end_time': u'2014-06-23 14:01:18.209680'
        }
    ]
}
```

List all projects

`ScrapydAPI.`**`list_projects`**`()`

Lists all available projects. See the list projects endpoint on Scrapyd's documentation.

Arguments:

- This method takes no arguments.

Returns: *(list)* A list of strings denoting the names of which projects are available.

```
>>> scrapyd.list_projects()
[u'ecom_project', u'estate_agent_project', u'car_project']
```

List all spiders in a project

`ScrapydAPI.`**`list_spiders`**(*project*)

Lists all spiders available to a given project. See the list spiders endpoint on Scrapyd's documentation.

Arguments:

- **project** *(string)* The name of the project to list spiders for.

Returns: *(list)* A list of strings denoting the names of spider available to the project.

```
>>> scrapyd.list_spiders('project_name')
[u'raw_spider', u'js_enhanced_spider', u'selenium_spider']
```

List all versions of a project

`ScrapydAPI.`**`list_versions`**(*project*)

This endpoint lists all available versions of a given project. See the list versions endpoint on Scrapyd's documentation.

Arguments:

- **project** *(string)* The name of the project to list versions for.

Returns: (list) A list of strings denoting all available version names for the requested project.

```
>>> scrapyd.list_versions('project_name'):
[u'345', u'346', u'347', u'348']
```

Schedule a job to run

`ScrapydAPI.`**`schedule`**(*project*, *spider*, *settings=None*, ***kwargs*)

The main action method which would actually cause scraping to start. This action schedules a given spider to run immediately if there are no concurrent jobs or as soon as possible once the current jobs are complete (this is a Scrapyd setting).

There is currently no built-in ability in Scrapyd to schedule a spider for a specific time, but this can be handled client side by simply firing off the request at the desired time.

See the schedule endpoint on Scrapyd's documentation.

Arguments:

- **project** *(string)* The name of the project that owns the spider.
- **spider** *(string)* The name of the spider you wish to run.
- **settings** *(dict)* A dictionary of Scrapy settings keys you wish to override for this run.
- **kwargs** Any extra parameters you would like to pass to the spiders constructor/init method.

Returns: (string) The Job ID of the newly created run.

```
# Schedule a job to run now sans extra parameters.
>>> scrapyd.schedule('project_name', 'spider_name')
u'14a6599ef67111e38a0e080027880ca6'
# Schedule a job to run now with overridden settings.
>>> settings = {'DOWNLOAD_DELAY': 2}
>>> scrapyd.schedule('project_name', 'spider_name', settings=settings)
u'23b5688df67111e38a0e080027880ca6'
```

```
# Schedule a job to run now with overridden settings.
# Schedule a joib to run now while passing init parameters.
>>> scrapyd.schedule('project_name', 'spider_name', extra_init_param='value')
u'14a6599ef67111e38a0e080027880ca6'
# Schedule a job to run now with overridden settings.
```

Note: 'project', 'spider' and 'settings' are reserved kwargs for this method and therefore these names should be avoided when trying to pass extra attributes to the spider init.

Handling Exceptions

As this library relies on the Requests library to handle HTTP connections, the exceptions raised by Requests itself for such things as hard connection errors, timeouts etc can be found in the Requests exceptions documentation.

However, when the problem is an error Scrapyd has returned itself instead, the `scrapyd_api.exceptions.ScrapydResponseError` will be raised with the applicable error message sent back from the Scrapyd API.

This works by simply checking the JSON return's *status* key and raising the exception with the return's *message* value, allowing the developer to debug the response.

A

C

D

J

L

S