OCDS Kingfisher Scrape

Feb 13, 2020

Contents

Kingfisher Scrape is a tool for downloading OCDS data and storing it on disk and/or sending it to an instance of Kingfisher Process for processing.

(If you are viewing this on GitHub, open the full documentation for additional details.)

You can:

- *Download data to your computer, by installing Kingfisher Scrape*
- *Download data to a remote server, by using Scrapyd*

You can also try using Kingfisher Scrape with Scrapy Cloud.

CHAPTER 1

How it works

Kingfisher Scrape is built on the Scrapy framework. Using this framework, we have authored "spiders" that you can run in order to "crawl" data sources and extract OCDS data.

When collecting data from a data source, each of its OCDS files will be written to a separate file on your computer. (Depending on the data source, an OCDS file might be a record package, release package, individual record or individual release.)

By default, these files are written to a `data` directory (you can *change this*) within your `kingfisher-scrape` directory (which you will create *during installation*). Each spider creates its own directory within the `data` directory, and each crawl of a given spider creates its own directory within its spider's directory. For example, if you run the `zambia` spider (*learn how*), then the directory hierarchy will look like:

```
kingfisher-scrape/
└── data
    └── zambia
        └── 20200102_030405
            ├── <...>.json
            ├── <...>.fileinfo
            └── <...>
```

As you can see, the `data` directory contains a `zambia` spider directory (matching the spider's name), which in turn contains a `20200102_030405` crawl directory (matching the time at which you started the crawl – in this case, 2020-01-02 03:04:05).

The crawl's directory will contain `.json` and `.fileinfo` files. The JSON files are the OCDS data. Each `.fileinfo` file contains metadata about a corresponding JSON file: the URL at which the JSON file was retrieved, along with other details.

1.1 Download data to your computer

This page will guide you through installing Kingfisher Scrape and using it to collect data from data sources.

1.1.1 Install Kingfisher Scrape

To use Kingfisher Scrape, you need access to a Unix-like shell (some are available for Windows). Git and Python (version 3.6 or greater) must be installed.

When ready, open a shell, and run:

```
git clone https://github.com/open-contracting/kingfisher-scrape.git
cd kingfisher-scrape
pip install -r requirements.txt
```

The next steps assume that you have changed to the `kingfisher-scrape` directory (the `cd` command above).

1.1.2 Configure Kingfisher Scrape

Note: This step is optional.

To use a different directory than the default `data` directory to store files, change the `FILES_STORE` variable in the `kingfisher_scrapy/settings.py` file. It can be a relative path (like `data`) or an absolute path (like `/home/user/path`).

```
FILES_STORE = '/home/user/path'
```

1.1.3 Collect data

You're now ready to collect data!

To list the spiders, run:

```
scrapy list
```

Alternately, you can get the list of spiders by accessing the `kingfisher_scrapy/spiders` directory in the GitHub repository. Each `.py` file is a spider, and the part before the `.py` extension is the spider's name.

The spiders' names might be ambiguous. If you're unsure which spider to run, you can compare their names to the list of OCDS publishers, or contact the OCDS Helpdesk at data@open-contracting.org.

To run a spider (that is, to start a "crawl"), replace `spider_name` below with the name of a spider from `scrapy list` above:

```
scrapy crawl spider_name
```

To download only a sample of the available data, add the `sample=true` spider argument:

```
scrapy crawl spider_name -a sample=true
```

Scrapy will then output a log of its activity.

Using an HTTP proxy

Note: This is an advanced topic. In most cases, you will not need to use this feature.

If the data source is blocking Scrapy's requests, you might need to use a proxy.

To use an HTTP and/or HTTPS proxy, add the `http_proxy` and/or `https_proxy` spider arguments:

```
scrapy crawl spider_name -a http_proxy=YOUR-PROXY-URL -a https_proxy=YOUR-PROXY-URL
```

1.1.4 Use data

You should now have a crawl directory within the `data` directory containing OCDS files. For help using data, read about using open contracting data.

1.2 Download data to a remote server

Note: This is an advanced guide that assumes knowledge of web hosting.

Some spiders take a long time to run (days or weeks), and some data sources have a lot of OCDS data (GBs). In such cases, you might not want to *download data to your computer*, and instead use a separate machine. You have two options:

1. Follow the same instructions as *before*, and start crawls on the other machine
2. Install Scrapyd on a remote server (this guide)

Scrapyd also makes it possible for many users to schedule crawls on the same machine.

1.2.1 Install Scrapyd

On the remote server, follow Scrapyd's installation instructions, then install the `requests` package in the same environment as Scrapyd:

```
pip install requests
```

1.2.2 Start Scrapyd

On the remote server, follow these instructions to start Scrapyd. Scrapyd should then be accessible at `http://your-remote-server:6800/`. If not, refer to Scrapyd's documentation or its GitHub issues to troubleshoot.

Using the Scrapyd web interface

- To see the scheduled, running and finished crawls, click "Jobs"
- To browse the crawls' log files, click "Logs"

For help understanding the log files, read :doc:crawl-report-guide'.

Note: If Scrapyd restarts or the server reboots, all scheduled crawls are cancelled, all running crawls are interrupted, and all finished crawls are delisted from the web interface. However, you can still browse the crawls' logs files.

1.2.3 Install Kingfisher Scrape

On your local machine, *install Kingfisher Scrape*.

1.2.4 Configure Kingfisher Scrape

Update the `url` variable in the `scrapy.cfg` file in your `kingfisher-scrape` directory, to point to the remote server. By default, the `scrapy.cfg` file contains:

```
[deploy]
url = http://localhost:6800/
project = kingfisher
```

You need to at least replace `localhost`. If you changed the `http_port` variable in Scrapyd's configuration file, you need to replace `6800`.

If you changed the `FILES_STORE` variable when *installing Kingfisher Scrape*, that same directory needs to exist on the remote server, and the `scrapyd` process needs permission to write to it. If you are using the default value, then files will be stored in a `data` directory under the Scrapyd directory on the remote server.

1.2.5 Deploy spiders

On your local machine, deploy the spiders in Kingfisher Scrape to Scrapyd, using the scrapyd-deploy command, which was installed with Kingfisher Scrape:

```
scrapyd-deploy
```

Remember to run this command every time you add or update a spider.

1.2.6 Collect data

Schedule a crawl, using Scrapyd's schedule.json API endpoint. For example, replace `localhost` with your remote server and `spider_name` with a spider's name:

```
curl http://localhost:6800/schedule.json -d project=kingfisher -d spider=spider_name
```

If successful, you'll see something like:

```
{"status": "ok", "jobid": "6487ec79947edab326d6db28a2d86511e8247444"}
```

Like when *downloading data to your computer*, you can download only a sample of the available data or *use a proxy* – just remember to use `-d` instead of `-a` before each spider argument. For example, replace `localhost` with your remote server and `spider_name` with a spider's name:

```
curl http://localhost:6800/schedule.json -d project=kingfisher -d spider=spider_name -
↪d sample=true
```

1.3 Crawl Report Guide

A Scrapy crawl is turned into a report. This page has tips on how to interpret this.

1.3.1 HTTP Response Codes

Look for a line that tells you how many 200 (Ok) response codes there were.

```
'downloader/response_status_count/200': 1,
```

Make sure there are no lines for HTTP codes that were not a 200 status. For example,

```
'downloader/response_status_count/404': 1,
```

Note there are times that spiders are able to recover from non-200 errors themselves.

For example, some of the Paraguay spiders need an authentication token. The server may send a 401 or 429 code if there are problems, and the spider can detect that and retry.

This means the presence of a non-200 line is not always a error, but it should always be checked.

1.3.2 Asking for help to interpret problems

Unfortunately, it's hard to give clear guides on how to interpret problems as the advice can differ a lot for different spiders.

For hosted Kingfisher, please ask a developer for help with any issues.

If you are running Kingfisher yourself, please open an issue in GitHub.

In both cases, we will try and build up this documentation as common patterns are uncovered.

1.4 Write spiders with Scrapy

1.4.1 About Scrapy

Scrapy calls scrapers 'spiders' so we're going to call them 'spiders' hereon.

Scrapy provides a *Spider* class which all spiders should inherit, and the process of writing a new spider basically involves defining the *parse* method and overriding any of the other *Spider* methods if you need to customise what they do.

The Scrapy framework determines what happens to a crawled item through its 'pipeline', and lots of things happen in the background (like the actual sending of requests, and automatic passing things through different stages in the pipeline).

Scrapy schedules HTTP requests with its own crawler engine, and they are not guaranteed (or likely) to be executed in a predictable order.

Scrapy documentation.

1.4.2 About OCDS Kingfisher

The OCDS data sources are for the most part JSON APIs which output valid OCDS data, either as `release_package`, `record_package`, `release`, `record` and a few other types, using various different structures or endpoints of the publishers' own design.

Our spiders need to:

- Know the URL(s) for a particular API from which all data can be found.

- **Tell the difference between endpoints which return lists of URLS**
  - And then follow these URLS
- **and endpoints which return OCDS data**
  - And then download this data.
- and endpoints which return both useful OCDS data and more URLs to fetch
- and endpoints which return other things.

This tends to involve prior knowledge of the API you're writing a spider for (you have to go look at its responses yourself to see what they are), and maybe some JSON parsing of the responses.

1.4.3 Using the pipeline

1. **Each spider first finds its starting point(s). Either:**
   - `start_urls` list or
   - requests yielded from the `start_requests` method.

Then, for *each* URL:

2. GET requests are passes it to the crawler engine to be executed. The response is passed back to the `parse` method in the spider.
3. The `parse` method must check the return status code! If it is not 200, this must be reported by fielding a block of information.
4. If the `parse` method has got a file it wants to save, it must call `save_response_to_disk` to do so! It should then yield a block of information.
5. The `parse` method can then yield further requests for processing.
6. The blocks of information are passed to the pipeline which fires it all off to the Kingfisher Process API.

The only parts you should have to touch when writing a spider are **1** and **3** to **5**.

Here is a sample:

```
class VerySimple(BaseSpider):
    name = "very_simple"

    def start_requests(self):
        # This API only has one URL to get. Make a request for that, and set a␣
↪filename
        yield scrapy.Request(
            url='https://buyandsell.gc.ca/cds/public/ocds/tpsgc-pwgsc_ocds_EF-FY-13-
↪14.json',
            meta={'kf_filename': '13-14.json'}
        )

    def parse(self, response):
        # We must check the response code
        if response.status == 200:
            # It was a success!
            # We must call to save to the disk
            self.save_response_to_disk(response, response.request.meta['kf_filename'])
            # We must send some information about this success
            yield {
```

(continues on next page)

(continued from previous page)

```
                'success': True,
                'file_name': response.request.meta['kf_filename'],
                "data_type": "release_package",
                "url": response.request.url,
            }
        else:
            # It was a failure :-(
            # We must send some information about this failure
            yield {
                'success': False,
                'file_name': response.request.meta['kf_filename'],
                "url": response.request.url,
                "errors": {"http_code": response.status}
            }
```

1.4.4 Spider properties

- `name`: a slug for the spider. This is what you pass to `scrapy crawl` to run it. Underscore separated, all lowercase. Required.
- `start_urls`: list of URLs to do the initial GET on. Don't need it if you define `start_requests` instead.
- See Scrapy Spider docs for other options.

```
from scrapy import Spider

class CanadaBuyAndSell(Spider):
    name = "canada_buyandsell"
    ...
```

1.4.5 Start Requests

Implement the `start_requests` method *instead of* using a `start_urls` property on the spider if you need to do something more complicated than just a list to get the URLs the spider starts with.

This might be useful to generate a long list of API endpoint URLs you know are sequential or contain dates or something.

However you come up with them, the output of this method should yield a Scrapy `Request` for each URL.

Eg.

```
def start_requests(self):
    url_base = 'https://buyandsell.gc.ca/cds/public/ocds/tpsgc-pwgsc_ocds_EF-FY-{}-{}.
↪json'
    urls = []
    for year in range(13, 17):
        urls.append(url_base.format(year, year+1))

    for url in urls:
        yield scrapy.Request(url)
```

This does the same thing as:

```
start_urls = [
        'https://buyandsell.gc.ca/cds/public/ocds/tpsgc-pwgsc_ocds_EF-FY-13-14.json',
        'https://buyandsell.gc.ca/cds/public/ocds/tpsgc-pwgsc_ocds_EF-FY-14-15.json',
        'https://buyandsell.gc.ca/cds/public/ocds/tpsgc-pwgsc_ocds_EF-FY-15-16.json',
        'https://buyandsell.gc.ca/cds/public/ocds/tpsgc-pwgsc_ocds_EF-FY-16-17.json',
    ]
```

Only with `start_requests` if we want to add a year we just up the range, or if the API endpoint changes we only need to modify one string.

1.4.6 Sample mode

Sample mode is a way to get a subset of the results, then stop the spider. It's triggered when you pass `-a sample=true` to `scrapy crawl <spider_name>`.

How sample mode is executed is different for every spider, depending on the API you're crawling. You *probably* want to define it in *start_requests* though, unless your `start_urls` is only one (like an index listing) in which case you'd define it in `parse` (where you loop through the listing).

It just needs to do something like yield a single Request for one URL in a list of URLs, instead of yielding Requests for all of the URLs in the list.

Eg. in `start_requests`:

```
if self.sample:
        yield scrapy.Request(urls[0])
    else:
        for url in urls:
            yield scrapy.Request(url)
```

Eg. in `parse`:

```
files_urls = json.loads(response.body)
    if self.sample:
        files_urls = [files_urls[0]]

    for file_url in files_urls:
        yield {
            'file_urls': [file_url],
            'data_type': 'record'
        }
```

1.5 Command-line tools

1.5.1 scrape-report

Extracts the Scrapy statistics from a crawl's log file.

```
python ocdskingfisher-scrape-cli scrape-report scrapyd/logs/kingfisher/spider_name/
↪6487ec79947edab326d6db28a2d86511.log
```

This is essentially the same as:

```
tac ../scrapyd/logs/kingfisher/spider_name/6487ec79947edab326d6db28a2d86511.log |␣
↪grep -B99 statscollectors | tac
```

1.5.2 log-dir-scrape-report

Extracts the Scrapy statistics from each crawl's log file, and writes them to a new `*_report.log` file.

```
python ocdskingfisher-scrape-cli log-dir-scrape-report scrapyd/logs/
```

This is essentially the same as:

```
find ../scrapyd/logs/ -type f -name "*.log" -not -name "*_report.log" -exec sh -c 'if␣
↪[ ! -f {}.stats ]; then result=$(tac {} | head -n99 | grep -m1 -B99 statscollectors␣
↪| tac); if [ ! -z "$result" ]; then echo "$result" > {}.stats; fi; fi' \;
```