Scrapy Streaming Documentation

Release 0.1a

Scrapy Developers and Contributors

Nov 08, 2017

Contents

CHAPTER 1

Scrapy Streaming

The Scrapy Streaming provides an interface to write spiders using any programming language, using json objects to make requests, parse web contents, get data, and more.

Also, we officially provide helper libraries to develop your spiders using Java, JS, and R.

1.1 Installation

Todo

Create pip package and document it here

1.2 Quickstart - Dmoz Streaming Spider

In this tutorial, we'll assume that both Scrapy and Scrapy Streaming is already installed on your system. If that's not the case, see *Installation*.

We'll develop the spider using Python because its simplicity, however, you can implement it using any programming language using the necessary syntax modifications.

In this tutorial, we'll implement a simple spider to extract data from http://www.dmoz.org.

1.2.1 Spider Behavior

If you are not familiar with Scrapy, we name Spider as an object that defines how scrapy should scrape information from a domain (or a group of domains). It contains all the logic and necessary information to extract the data from a website.

We'll define a simple spider, that works as follows:

- The spider initializes at http://www.dmoz.org/Computers/Programming/Languages/Python/
- For each subcategory in `Subcategories` section, we need to open its page, and extract the link of external websites in the `Sites` section.
- Extracted values will be saved in a dictionary that maps `title: url`.

1.2.2 Scrapy Streaming

The Scrapy Streaming plugin uses the default `stdin` and `stdout` to communicate with Scrapy. This means that you must have Scrapy installed on your system to use it with the plugin.

Scrapy will be responsible for starting your process (defined as external spider) and create a communication channel that uses json messages.

Therefore, the spider can be implemented in any programming language as long as it supports communication with the system `stdin` and `stdout`.

1.2.3 Dmoz Spider Implementation

Now we'll implement the sample spider using Python language. Read the *Communication Protocol* for more information about it.

Each message ends with a line break `\n`, so it's important to read and send a single message per line.

After starting your process, Scrapy'll let it know that the communication channel is ready sending the following message:

```
{
    "type": "status",
    "status": "ready"
}
```

So, the first thing is to wait the scrapy's confirmation. We start implementing as follows:

Listing 1.1: dmoz_spider.py

```
#! /usr/bin/env python
import json
from sys import stdin, stdout
from scrapy.selector import Selector


def parse_json(line):
    # parses the line string to a python object
    return json.loads(line)


def write_line(data):
    # converts data to a single line message and write it in the system stdout
    msg = ''.join([line.strip() for line in data.splitlines()])
    stdout.write(msg + '\n')

pending_requests = 0
result = {}

def main():
    status = parse_json(stdin.readline())
```

```
22
23      # we start checking if the channel is ready
24      if status['status'] != 'ready':
25          raise Exception("There is problem in the communication channel")
26
27      # continue with the implementation here
28
29  if __name__ == '__main__':
30      main()
```

We read the line from system `stdin` and confirms that it's a `status` ready confirmation.

The code above defines two helper functions, `parse_json` that receives an string and convert it to a python object (a dict); and `write_line` that receives a multiline string and convert it to a single-line one, and write it to the `stdout` with a line-break.

Now, we must provide the *spider* information. On line `27`, a spider is defined adding the following code:

```
write_line('''
    {
        "type": "spider",
        "name": "dmoz",
        "start_urls": ["http://www.dmoz.org/Computers/Programming/Languages/Python/"]
    }
''')
```

With this message, the scrapy steaming will create a Spider and start its execution, requesting the `start_urls` pages.

After the `write_line` call, we implement a loop that will be always checking the system `stdin`. This loop will check if the spider got some problems in the execution, and analyze the responses.

We define the main loop as:

```
while True:
    msg = parse_json(stdin.readline())

    # check the message type
    if msg['type'] == 'exception' or msg['type'] == 'error':
        raise Exception("Something wrong... " + str(msg))

    elif msg['type'] == 'response':
        # we check the id of the incoming response, and call a function to extract
        # the data from each page
        if msg['id'] == 'parse':
            response_parse(msg)
        elif msg['id'] == 'category':
            response_category(msg)
```

The code above start checking if there is some problem in the spider, and then check it's a response.

Our spider will have two type of responses:

- **parse**: this is sent after receiving the content from `start_urls`
- **category**: this is sent after receiving the content of each subcategory (we'll implement it soon)

Responses that has the `id` field equals to `parse` comes from the `start_urls` requests. So, let's start implementing the `response_parse` method. This method will get a list of subcategories at http://www.dmoz.org/Computers/Programming/Languages/Python/ and open a new request to each subcategory page.

Let's implement the `response_parse` function. This function receives the response from the initial url and open a new request to each repository.

```
def response_parse(response):
    global pending_requests
    # using scrapy selector to extract data from the html
    selector = Selector(text=response['body'])
    # get the url of subcategories
    for href in selector.css("#subcategories-div > section > div > div.cat-item >␣
↪a::attr('href')"):
        # we count the number of requests using this var
        pending_requests += 1
        # open a new request
        write_line('''
            {
                "type": "request",
                "id": "category",
                "url": "http://www.dmoz.org%s"
            }
        ''' % href.extract())
```

We are using scrapy's Selector to extract data from the html body, but feel free to use anyone. For each subcategory html, we open a new request using the write_line with the *request* message. Notice that these requests are using the `id` equals to `category`, so its responses will have a field with the same value.

Finally, let's implement the `response_category` method. This method receives the response of each subcategory.

```
def response_category(response):
    global pending_requests
    # this response is no longer pending
    pending_requests -= 1

    # using scrapy selector
    selector = Selector(text=response['body'])
    # get div with link and title
    divs = selector.css('div.title-and-desc')

    for div in divs:
        url = div.css("a::attr('href')").extract_first();
        title = div.css("a > div.site-title::text").extract_first();
        result[title] = url

    # if finished all requests, we can close the spider
    if pending_requests == 0:
        # serialize the extracted data and close the spider
        open('outputs/dmoz_data.json', 'w').write(json.dumps(result))
        write_line('{"type": "close"}')
```

For each response received, we decrease the `pending_requests` value, and the we close the spider when there is no pending request.

Now, to run your spider use the following command:

```
scrapy streaming dmoz_spider.py
```

This command will start your process and run your spider until receive the *close* message.

1.2.4 Source code

The source used in this section:

```
#! /usr/bin/env python
import json
from sys import stdin, stdout
from scrapy.selector import Selector


def parse_json(line):
    # parses the line string to a python object
    return json.loads(line)


def write_line(data):
    # converts data to a single line message and write it in the system stdout
    msg = ''.join([line.strip() for line in data.splitlines()])
    stdout.write(msg + '\n')
    stdout.flush()

pending_requests = 0
result = {}


def response_parse(response):
    global pending_requests
    # using scrapy selector to extract data from the html
    selector = Selector(text=response['body'])
    # get the url of repositories
    for href in selector.css("#subcategories-div > section > div > div.cat-item >␣
↪a::attr('href')"):
        # we count the number of requests using this var
        pending_requests += 1
        # open a new request
        write_line('''
            {
                "type": "request",
                "id": "category",
                "url": "http://www.dmoz.org%s"
            }
        ''' % href.extract())


def response_category(response):
    global pending_requests
    # this response is no longer pending
    pending_requests -= 1

    # using scrapy selector
    selector = Selector(text=response['body'])
    # get div with link and title
    divs = selector.css('div.title-and-desc')

    for div in divs:
        url = div.css("a::attr('href')").extract_first();
        title = div.css("a > div.site-title::text").extract_first();
        result[title] = url
```

```
54
55      # if finished all requests, we can close the spider
56      if pending_requests == 0:
57          # serialize the extracted data and close the spider
58          open('outputs/dmoz_data.json', 'w').write(json.dumps(result))
59          write_line('{"type": "close"}')
60
61
62  def main():
63      status = parse_json(stdin.readline())
64
65      # we start checking if the channel is ready
66      if status['status'] != 'ready':
67          raise Exception("There is problem in the communication channel")
68
69      write_line('''
70          {
71              "type": "spider",
72              "name": "dmoz",
73              "start_urls": ["http://www.dmoz.org/Computers/Programming/Languages/
    ↪Python/"]
74          }
75      ''')
76
77      while True:
78          msg = parse_json(stdin.readline())
79
80          # check the message type
81          if msg['type'] == 'exception' or msg['type'] == 'error':
82              raise Exception("Something wrong... " + str(msg))
83
84          elif msg['type'] == 'response':
85              # we check the id of the incoming response, and call a funtion to extract
86              # the data from each page
87              if msg['id'] == 'parse':
88                  response_parse(msg)
89              elif msg['id'] == 'category':
90                  response_category(msg)
91
92
93  if __name__ == '__main__':
94      main()
```

1.3 External Spiders

We define `External Spider` as a spider developed in any programming language.

The external spider must communicate using the system `stdin` and `stdout` with the Streaming.

1.3.1 Stand-alone spiders

You can run standalone external spiders using the `streaming` command.

streaming

- Syntax: `scrapy streaming [options] <path of executable>`
- Requires project: *no*

and if you need to use extra arguments, add them using the `-a` parameter:

```
scrapy streaming my_executable -a arg1 -a arg2 -a arg3,arg4
```

So, to execute a java spider named `MySpider` for example, you may use:

```
scrapy streaming java -a MySpider
```

1.3.2 Integrate with Scrapy projects

If you want to integrate external spiders with a scrapy's project, create a file named `external.json` in your project root. This file must contain an array of json objects, each object with the `name` , `command`, and `args` attributes.

To use a different directory, instead of the project root, you can set the `EXTERNAL_SPIDERS_PATH` setting with a absolute directory. If the setting `EXTERNAL_SPIDERS_PATH` is set, Scrapy Streaming will use the spiders defined at `EXTERNAL_SPIDERS_PATH/external.json`.

The `name` attribute will be used as documented in the `Spider.name`. The `command` is the path or name of the executable to run your spider. The `args` attribute is optional, this is an array with extra arguments to the command, if any.

For example, if you want to add spider developed in Java and a binary spider, you can define the `external.json` as follows:

```
[
  {
    "name": "java_spider",
    "command": "java",
    "args": ["/home/user/MySpider"]
  },
  {
    "name": "compiled_spider",
    "command": "/home/user/my_executable"
  }
]
```

You can execute external spiders using the `crawl` command.

crawl

- Syntax: `scrapy crawl <spider_name>`
- Requires project: *yes*

This command starts a spider given its name.

Usage examples:

```
scrapy crawl java_spider
```

1.4 Communication Protocol

The communication between the external spider and Scrapy Streaming will use json messages.

Every message must be escaped and ends with a line break `\n`. Make sure to flush the stdout after sending a message to avoid buffering.

After starting the spider, Scrapy will let it know that the communication chanel is ready sending the *ready* message.

Then, as the first message, the spider must send a *spider* message with the necessary information. The Scrapy Streaming will start the spider execution and return the *response* with `id` equals to `parse`. Following this message, the external spider can sends any message listed bellow. To finish the spider execution, it must send the *close* message.

The implementation of such procedure should contain a main loop, that checks the `type` of the data received from Streaming, and then new actions can be done. It's recommended to always check if the message if an *exception* to avoid bugs in your implementation. Also, if the Streaming finds anything wrong with the json message sent by the spider, a *error* message with the necessary information will be sent.

Tip: Every message contains a field named `type`. You can use this field to easily identify the incoming data. The possible types are listed bellow.

Scrapy Streaming Messages:

- *ready*
- *response*
- *response_selector*
- *exception*
- *error*

External Spider Messages:

- *spider*
- `log`
- *request*
- *from_response_request*
- *selector_request*
- *close*

Note: In this documentation, we use the $\star$ to identify that a field is optional. When implementing your spider, you can omit this field and you must NOT use the $\star$ character in the field name as described here.

1.4.1 ready

This message is sent by Streaming after starting and connecting with the process stdin/stdout. This is a confirmation that communication channel is working.

```
{
    "type": "ready",
    "status": "ready"
}
```

1.4.2 response

Scrapy Streaming will serialize part of the `Response` object. See `Response` for more information.

The response `id` will be the same that used in the *`request`*. If it's the response from the initial spider urls, the request `id` will be `parse`.

```
{
    "type": "response",
    "id": string,
    "url": string,
    "headers": object,
    "status": int,
    "body": string,
    "meta": object,
    "flags": array
}
```

1.4.3 response_selector

This message will be sent by Streaming after receiving the response from a *`selector_request`*.

It contains the fields as described in *`response`*, plus an additional `selector` field that is a dictionary mapping from a field name to an array of extracted data.

```
{
    "type": "response_selector",
    // ..., all response fields
    "selector": object mapping field name to an array strings with extracted data
}
```

1.4.4 exception

Exceptions are thrown when Scrapy faces a runtime error, such as requesting an invalid domain, being unable to find a form in a `form_request`, etc.

Each *`exception`* contains the received message that caused this error, and the exception's message.

```
{
    "type": "exception",
    "received_message": string,
    "exception": string
}
```

1.4.5 error

Errors are thrown if there is any problem with the validation of the received message. Runtime errors are thrown by *exception*.

If the Spider is using an unknown type, or an invalid field, for example, this message will be sent with the necessary information.

The Streaming will send the error details, and stops its execution.

The *error* contains `received_message` field with the message received from external spider that generated this error and `details` field, with a hint about what may be wrong with the spider.

```
{
    "type": "error",
    "received_message": string,
    "details": string
}
```

1.4.6 spider

This is the first message sent by your spider to Scrapy Streaming. It contains information about your Spider. Read the `Spider` docs for more information.

```
{
    "type": "spider",
    "name": string,
    "start_urls": array,
    *"allowed_domains": array,
    *"custom_settings": object
}
```

1.4.7 log

Log message allows the external spider to add log messages in the scrapy streaming output. This may be helpful in the spider development to track variables, responses, etc.

The log message is defined as follows:

```
{
    "type": "log",
    "message": string,
    "level": string
}
```

The message level must be one of the following:

- `CRITICAL` - for critical errors (highest severity)
- `ERROR` - for regular errors
- `WARNING` - for warning messages
- `INFO` - for informational messages
- `DEBUG` - for debugging messages (lowest severity)

1.4.8 request

To open new requests in the running spider, use the request message. Read the `Request` for more information.

The *`request`* must contains the `id` field. Scrapy Streaming will send the response with this same `id`, so each response can be easily identified by its id.

```
{
    "type": "request",
    "id": string,
    "url": string,
    *"base64": bool,
    *"method": string,
    *"meta": object,
    *"body": string,
    *"headers": object,
    *"cookies": object or array of objects,
    *"encoding": string,
    *"priority": int,
    *"dont_filter": boolean
}
```

If the `base64` parameter is `true`, the response body will be encoded using base64.

Note: Binary responses, such as files, images, videos, etc, must be encoded with base64. Therefore, when using scrapy-streaming to download binary data, you **must** set the `base64` parameter to `true` and decode the response's body with the base64 encoding.

1.4.9 from_response_request

The *`from_response_request`* uses the `from_response()` method. Check the `FormRequest` for more information.

It first creates a `Request` and then use the response to create the `FormRequest`

The type of this message is *`from_response_request`*, it contains all fields described in *`request`* doc, and the `from_response()` data in the `from_response_request` field.

You can define it as follows:

```
{
    "type": "from_response_request",

    ... // all request's fields here

    "from_response_request": {
        *"formname": string,
        *"formxpath": string,
        *"formcss": string,
        *"formnumber": int,
        *"formdata": object,
        *"clickdata": object,
        *"dont_click": boolean
    }
}
```

The *from_response_request* will return the response obtained from FormRequest if successful.

1.4.10 selector_request

The *selector_request* can be used in order to extract items using multiple selectors.

It first creates a Request and then parses the result with the desired selectors.

The type of this message is *selector_request*, it contains all fields described in *request*, and the selector object with the item fields and its corresponding selectors.

```
{
    "type": "item_selector_request",
    ... // all request's fields here

    "selector": {
        "field 1": {
            "type": "css" or "xpath",
            "filter": string
        },
        "field 2": {
            "type": "css" or "xpath",
            "filter": string
        }

        ... // use field name: selector object
    }
}
```

Each key of the selector object is the field name, and its value is a selector.

The *selector_request* will return a list with the extracted items if successful. Each item will be an object with its fields and extracted values, defined as follows:

```
{
    "type": "response_selector",
    ... // all response's fields here

    "selector": {
        "field 1": ['item 1', 'item 2', ...],
        "field 2": ['item 1', 'item 2', 'item 3', ...]
    }
}
```

1.4.11 close

To finish the spider execution, send the *close* message. It'll stop any pending request, close the communication channel, and stop the spider process.

The *close* message contains only the type field, as follows:

```
{
    "type": "close"
}
```

1.5 Spider Examples

We provide some simple examples to demonstrate how to use Scrapy Streaming features.

You can download these examples from https://github.com/aron-bordin/scrapy-streaming/tree/examples/examples

Todo

Update the examples URL to master

1.5.1 Examples

1. **check_response_status** - This spider open a list of domains and check which domain is returning a valid status.
2. **extract_dmoz_links** - This example is covered in the quickstart section. It gets a list of websites with Python related articles
3. **request_image** - This demo shows how to download binary data.
4. **request_utf8** - Shows that Scrapy Streaming supports UTF8 encoding.
5. **fill_form** - This example covers the `from_response_request` to fill a form with some data.
6. **post_request** - This example shows you how to do a POST request.

Example	Python	R	Java	Node.js	Post	More
1	OK	OK	OK	OK	OK	x
2	OK	OK	OK	OK	OK	x
3	OK	OK	OK	OK	OK	x
4	OK	x	OK	OK	OK	x
5	OK	OK	OK	OK	OK	x

- The Python examples are using the raw *Communication Protocol*, sending json strings in the stdout. It's recommended to follow theses examples if you are seeking a better understanding of the Scrapy Streaming behavior.
- R examples are using the `scrapystreaming` package, you can read the documentation here: *R Package*.
- Java examples are using the `scrapystreaming` library, you can read the documentation here: *Java Library*
- Node.js examples are using the `scrapystreaming` package, you can read the documentation here: *Node.js Package*

1.6 Java Library

Todo

publish it and add the link to maven central repository

We provide a helper library to help the development process of external spiders using Java.

It's recommended to read the *Quickstart - Dmoz Streaming Spider* before using this package.

1.6.1 Installation

Todo

requires maven artifact

1.6.2 Gson

`scrapystreaming` depends on Gson. It's used to manipulate json data with Java.

> Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson can work with arbitrary Java objects including pre-existing objects that you do not have source-code of.

We'll present the following Gson's methods:

- *toJson()*
- *fromJson()*

These commands may be helpful in the spider development.

You can read the official user guide here: https://github.com/google/gson/blob/master/UserGuide.md.

toJson (*Object object*)

The *toJson()* allows you to convert a Java object to JSON, and can be used as follows:

```
Gson gson = new Gson();
gson.toJson(1);             // ==> 1
gson.toJson("abcd");        // ==> "abcd"
gson.toJson(new Long(10)); // ==> 10
int[] values = { 1 };
gson.toJson(values);        // ==> [1]
```

fromJson (*String json*, *Class klass*)

You can use the *fromJson()* command to converts a json string to a Java object again.

```
int one = gson.fromJson("1", int.class);
Integer one = gson.fromJson("1", Integer.class);
Long one = gson.fromJson("1", Long.class);
Boolean false = gson.fromJson("false", Boolean.class);
String str = gson.fromJson("\"abc\"", String.class);
String[] anotherStr = gson.fromJson("[\"abc\"]", String[].class);
```

You can read more advanced examples in the Gson user guide: https://github.com/google/gson/blob/master/UserGuide.md.

If you prefer, you can access a gson instance from `org.scrapy.scrapystreaming.utils.Utils.gson`.

1.6.3 scrapystreaming

The `scrapystreaming` library provide the following objects:

- *Spider*
- *Callback*
- *SpiderException*

- *Request*
- *FromResponseMessage*
- *FromResponseRequest*
- *Logger*

class Spider
(org.scrapy.scrapystreaming.Spider)

This class defines your spider.

To implement a Spider, extends this class and implements the *parse()* method.

name
(String) The name of the Spider, defaults to `ExternalSpider`.

start_urls
(List<String>) Initial URLs, defaults to an empty array list.

allowed_domains
(List<String>) Allowed domains, defaults to null.

custom_settings
(Map) Spider custom settings, defaults to null.

start()
(void) Initializes the Spider execution. Throws *SpiderException*.

close()
(void) Sends a *close* message to stops the Spider execution.

parse(*ResponseMessage response*)
(abstract void) This method must be implemented to parse the response from initial URLs.

onException(*ExceptionMessage exception*)
(void) This method is called when Scrapy raises an exception and sends the exception message. If you want to analyze the exception, or just ignore the problem, override this function.

Throws *SpiderException*.

class Callback
(org.scrapy.scrapystreaming.core.Callback)

Callback is an interface to handle responses. The *Spider* implements this interface, and you need to provide a callback instance to open new requests.

parse(*ResponseMessage response*)
(void) Method to handle to response content

class SpiderException
(org.scrapy.scrapystreaming.core.SpiderException)

Exceptions raised by Scrapy Streaming and Scrapy

class Request
(org.scrapy.scrapystreaming.Request)

Class responsible for creating new requests.

Request(*String url*)
Creates the Request object with a given URL

open (*Callback callback*)
(void) Opens the request, and parse the response in the callback instance.

The callback can be any class that implements the *Callback* interface, including the spider instance.

Throws *SpiderException*.

class FromResponseMessage
(org.scrapy.scrapystreaming.messages.FromResponseMessage)

This is a simple class to put extra data to *FromResponseRequest* requests.

formname
(String) FromResponseRequest's formname parameter

formxpath
(String) FromResponseRequest's formxpath parameter

formcss
(String) FromResponseRequest's formcss parameter

formnumber
(Integer) FromResponseRequest's formnumber parameter

formdata
(Map) FromResponseRequest's formdata parameter

clickdata
(Map) FromResponseRequest's clickdata parameter

dont_click
(Boolean) FromResponseRequest's dont_click parameter

class FromResponseRequest
(org.scrapy.scrapystreaming.Request)

Class responsible for creating new requests using response and extra data.

FromResponseRequest (*String url*, *FromResponseMessage from_response_request*)
Creates the FromResponseRequest object with a given URL, using the *FromResponseMessage* data.

open (*Callback callback*)
(void) Opens the request, and parse the response in the callback instance.

The callback can be any class that implements the *Callback* interface, including the spider instance.

Throws *SpiderException*.

class Logger
(org.scrapy.scrapystreaming.Logger)

The Logger class lets you write log messages in the Scrapy Streaming logger.

classmethod log (*String message*, *LEVEL level*)
(void) Write a log message.

LEVEL is a enum defined in `Logger.LEVEL`, and can be CRITICAL, ERROR, WARNING, INFO, or DEBUG.

classmethod critical (*String message*)
(void) Write a critical message in the Scrapy Streaming logger.

classmethod error (*String message*)
(void) Write a error message in the Scrapy Streaming logger.

classmethod `warning` (*String message*)
(void) Write a warning message in the Scrapy Streaming logger.

classmethod `info` (*String message*)
(void) Write a info message in the Scrapy Streaming logger.

classmethod `debug` (*String message*)
(void) Write a debug message in the Scrapy Streaming logger.

1.6.4 Dmoz Streaming Spider with Java

In this section, we'll implement the same spider developed in *Quickstart - Dmoz Streaming Spider* using the `scrapystreaming` java library. It's recommended that you have read the quickstart section before following this topic, to get more details about Scrapy Streaming and the spider being developed.

We'll be using the jsoup selector to analyze the html content, feel free to use any one.

We start by defining the Spider class and creating two variables:

```
public class Dmoz extends Spider {
    // we use the numRequests to count remaining requests
    static int numRequests = 0;
    // the results variable store the extracted data, mapping from title: url
    static HashMap<String, String> results = new HashMap<String, String>(0);

    Dmoz() {
        name = "dmoz";
        // set the initial url
        start_urls.add("http://www.dmoz.org/Computers/Programming/Languages/Python/");
    }
```

Then, we must implement the *`Spider.parse()`* method to handle the response from start_urls requests:

```
public void parse(ResponseMessage response) {
    // get the initial page, and open a new request to each subcategory
    Document doc = Jsoup.parse(response.body);
    Elements hrefs = doc.select("#subcategories-div > section > div > div.cat-item >␣
↪a[href]");
    for (Element el: hrefs) {
        try {
            Request r = new Request("http://www.dmoz.org" + el.attr("href"));
            r.open(new Callback() {
                public void parse(ResponseMessage response) {
                    parseSubcat(response);
                }
            });

            // increments the number of open requests
            numRequests++;
        } catch (SpiderException e) {
            e.printStackTrace();
        }
    }
}
```

This method get external links usign the jsoup selector, and increment the `numRequests` variables, so we can keep trace of the number of webpages being extracted.

We create a new request to each link found, and the get the response using the *Callback* object, and the calling the `parseSubcat` method.

Tip: If instead of creating a new *Callback* object to each request, you can pass a single object that implements the callback interface, or just pass the spider instance.

Now, let's implement the `parseSubcat` method.

```
public void parseSubcat(ResponseMessage response) {
    // decrement the number of open requests
    numRequests--;
    Document doc = Jsoup.parse(response.body);
    Elements divs = doc.select("div.title-and-desc a");

    // extract all urls in the page
    for (Element item: divs) {
        String url = item.attr("href");
        String title = item.select("div.site-title").first().text();
        results.put(title, url);
    }

    // close the spider and save the data, when necessary
    if (numRequests == 0) {
        try {
            Writer writer = new FileWriter("outputs/dmoz.json");
            Utils.gson.toJson(results, writer);
            writer.flush();
        } catch (Exception e) {
            e.printStackTrace();
        }
        close();
    }
}
```

The `parseSubcat` method first extract the external link title and href, and then adds it to the results variable.

When there is no response remaining (numRequests is equal to 0), the extracted data is converted to json and written in the disk.

Following that, the *Spider.close()* is called, to close the spider.

Finally, we add a main method to make this spider executable.

```
public static void main(String args[]) throws Exception {
    Dmoz spider = new Dmoz();
    spider.start();
}
```

To execute the spider, you can use something similar to:

```
scrapy streaming java -a -cp,.:\*,Dmoz
```

Where `-a` means that we are adding some arguments to the java command. The arguments used here are:

- -cp,.:* to add the required .jars in the java classpath (if you have both scrapystreaming and its dependencies in the java classpath, you can skip this parameter)
- a comma, to separate the next argument

- Dmoz, the name of the class

1.7 R Package

Todo

publish it and add the link to cran package with the package docs here

We provide a helper library to help the development process of external spiders in R.

It's recommended to read the *Quickstart - Dmoz Streaming Spider* before using this package.

1.7.1 Installation

To install the `scrapystreaming` package, runs:

```
install.packages("scrapystreaming")
```

and loads it using:

```
library(scrapystreaming)
```

1.7.2 jsonlite

`scrapystreaming` will load the jsonlite package.

> The jsonlite package offers flexible, robust, high performance tools for working with JSON in R and is particularly powerful for building pipelines and interacting with a web API.

We focus our attention to two jsonlite's commands:

- *toJSON()*
- *fromJSON()*

These commands may be helpful in the spider development.

You can read the jsonlite's quickstart here: https://cran.r-project.org/web/packages/jsonlite/vignettes/json-aaquickstart.html and a more detailed documentation here: http://arxiv.org/abs/1403.2805

toJSON(*x*, *dataframe = c("rows", "columns", "values")*, *matrix = c("rowmajor", "columnmajor")*, *Date = c("ISO8601", "epoch")*, *POSIXt = c("string", "ISO8601", "epoch", "mongo")*, *factor = c("string", "integer")*, *complex = c("string", "list")*, *raw = c("base64", "hex", "mongo")*, *null = c("list", "null")*, *na = c("null", "string")*, *auto_unbox = FALSE*, *digits = 4*, *pretty = FALSE*, *force = FALSE*, ...)

The *toJSON()* allows you to convert a R object to JSON, and can be used as follows:

```
> test1 <- data.frame(a = "a field", b = 2)
> toJSON(test1)
[{"a":"a field","b":2}]
>
> nested_data <- data.frame(user = "admin", pass = "secret")
> data <- data.frame(login = "ok", attempts = 2, input = NA)
> data$input <- nested_data
```

```
> toJSON(data)
[{"login":"ok","attempts":2,"input":{"user":"admin","pass":"secret"}}]
```

fromJSON (*txt*, *simplifyVector = TRUE*, *simplifyDataFrame = simplifyVector*, *simplifyMatrix = simplifyVector*, *flatten = FALSE*, ...)

You can use the *fromJSON()* command to converts a json string to a R object again.

```
> converted_test1 <- fromJSON('[{"a":"a field","b":2}]')
> converted_data <- fromJSON('[{"login":"ok","attempts":2,"input":{"user":"admin",
↪"pass":"secret"}}]')
```

1.7.3 scrapystreaming

The `scrapystreaming` R package provide the following commands:

- *create_spider()*
- *close_spider()*
- *send_log()*
- *send_request()*
- *send_from_response_request()*
- *parse_input()*
- *handle()*
- *run_spider()*

Tip: The Scrapy Streaming and your spider communicates using the system `stdin`, `stdout`, and `stderr`. So, don't write any data that is not a json message to the system `stdout` or `stderr`.

These commands write and read data from `stdin`, `stdout`, and `stderr` when necessary, so you don't need to handle the communication channel manually.

---

create_spider (*name*, *start_urls*[, *callback*, *allowed_domains*, *custom_settings*])

Parameters

- **name** (*character*) – The name of the spider
- **start_urls** (*character*) – vector with initial urls
- **callback** (*function*) – the function to handle the response callback from start_urls requests, must receive one parameter `response` that is a data.frame with the response data
- **allowed_domains** (*character*) – vector with allowed domains (optional)
- **custom_settings** (*data.frame*) – custom scrapy settings (optional)

This command is used to create and run a Spider, sending the *spider* message.

Usages:

```
parse <- function (response) {
    # do something with response
}
```

```
# usage 1, all parameters
create_spider(name = "sample",
              start_urls = "http://example.com",
              allowed_domains = c("example.com", "dmoz.org"),
              callback = parse,
              custom_settings = data.frame("SOME_SETTING" = "some value"))

# usage 2, empty spider
create_spider(name = "sample", start_urls = character(0))
```

close_spider()

Closes the spider, sending the *close* message.

Usage:

```
close_spider()
```

send_log(*message*, *level = "DEBUG"*)

Parameters

- **message** (*character*) – the log message
- **level** (*character*) – log level. Must be one of 'CRITICAL', 'ERROR', 'WARNING', 'DEBUG', and 'INFO', defaults to DEBUG

Send a log message to the Scrapy Streaming's logger output. This commands sends the `log` message.

Usages:

```
# default DEBUG message
send_log("starting spider")

# logging some error
send_log("something wrong", "error")
```

send_request(*url*, *callback*[, *base64*, *method*, *meta*, *body*, *headers*, *cookies*, *encoding*, *priority*, *dont_filter*])

Parameters

- **url** (*character*) – request url
- **callback** (*function*) – the function to handle the response callback, must receive one parameter `response` that is a data.frame with the response data
- **base64** (*logical*) – if TRUE, the response body will be encoded with base64 (optional)
- **method** (*character*) – request method (optional)
- **meta** (*data.frame*) – metadata to the request (optional)
- **body** (*character*) – the request body (optional)
- **headers** (*data.frame*) – request headers (optional)
- **cookies** (*data.frame*) – request cookies (optional)
- **encoding** (*character*) – request encoding (optional)
- **priority** (*numeric*) – request priority (optional)
- **dont_filter** (*logical*) – indicates that this request should not be filtered by the scheduler (optional)

Open a new request, using the *request* message.

Usages:

```
my_callback <- function (response) {
    # do something with response
}

# simple request
send_request("http://example.com", my_callback)

# base64 encoding, used to request binary content, such as files
send_request("http://example.com/some_file.xyz", my_callback, base64 = TRUE)
```

send_from_response_request (*url*, *callback*, *from_response_request*[, *base64*, *method*, *meta*, *body*, *headers*, *cookies*, *encoding*, *priority*, *dont_filter*])

Parameters

- **url** (*character*) – request url
- **callback** (*function*) – the function to handle the response callback, must receive one parameter `response` that is a data.frame with the response data
- **from_response_request** (*data.frame*) – data to the new request. May contain fields to the request and to the form.
- **from_response_request$method** (*character*) – request method (optional)
- **from_response_request$meta** (*data.frame*) – metadata to the request (optional)
- **from_response_request$body** (*character*) – the request body (optional)
- **from_response_request$headers** (*data.frame*) – request headers (optional)
- **from_response_request$cookies** (*data.frame*) – request cookies (optional)
- **from_response_request$encoding** (*character*) – request encoding (optional)
- **from_response_request$priority** (*numeric*) – request priority (optional)
- **from_response_request$dont_filter** (*logical*) – indicates that this request should not be filtered by the scheduler (optional)
- **base64** (*logical*) – if TRUE, the response body will be encoded with base64 (optional)
- **method** (*character*) – request method (optional)
- **meta** (*data.frame*) – metadata to the request (optional)
- **body** (*character*) – the request body (optional)
- **headers** (*data.frame*) – request headers (optional)
- **cookies** (*data.frame*) – request cookies (optional)
- **encoding** (*character*) – request encoding (optional)
- **priority** (*numeric*) – request priority (optional)
- **dont_filter** (*logical*) – indicates that this request should not be filtered by the scheduler (optional)

This function creates a request, and then use its response to open a new request using the *from_response_request* message.

Usages:

```
my_callback <- function (response) {
    # do something with response
}

# submit a login form, first requesting the login page, and then submitting the form

# we first create the form data to be sent
from_response_request <- data.frame(formcss = character(1), formdata = character(1))
from_response_request$formcss <- "#login_form"
from_response_request$formdata <- data.frame(user = "admin", pass = "1")

# and open the request
send_from_response_request("http://example.com/login", my_callback, from_response_
↪request)
```

parse_input (*raw = FALSE, ...*)

Parameters

- **raw** (*logical*) – if TRUE, returns the raw input line. If FALSE (the default), parse it using jsonlite::fromJSON (optional)
- **..** – extra arguments to jsonlite::fromJSON (optional)

This method reads the `stdin` data.

If raw is equal to TRUE, returns the string of one line received.

Otherwise, it will parse the line using *fromJSON()*, with ... as argument.

Usages:

```
# raw message
line <- parse_input(raw = TRUE)

# json message
msg <- parse_input()

# message with custom toJSON parameters
msg <- parse_input(FALSE, simplifyVector = FALSE, simplifyDataFrame = FALSE)
```

handle (*execute = TRUE*)

Parameters **execute** (*logical*) – if TRUE (the default) will execute the message as follows:

- ready: if the status is ready, continue the execution
- response: call the callback with the message as the first parameter
- error: stop the execution of the process and show the spider error
- exception: stop the execution of the process and show the scrapy exception

Read a line from `stdin` and processes it.

if false, will return the line processed with *fromJSON()*.

Usages:

```
# usage 1, keep executing incoming data
while (TRUE) {
    handle()
```

```
}

# usage 2, handle a single message
# first, open the request
send_request("http://example.com", my_callback)
# and then processes the stdin to get the response, some exception, or errors
handle()
```

run_spider()

This method is a shortcut to always handle the incoming data. It's a simple loop that keep using the *handle()*

Usage:

```
# create the spider

...
create_spider("sample", c("url1", "url2", ...), parse)
# and then keep always processing the scrapy streaming messages in the process stdin
run_spider()
```

The *run_spider()* is equivalent to

```
while (TRUE) {
    handle()
}
```

1.7.4 Dmoz Streaming Spider with R

In this section, we'll implement the same spider developed in *Quickstart - Dmoz Streaming Spider* using the `scrapystreaming` package. It's recommended that you have read the quickstart section before following this topic, to get more details about Scrapy Streaming and the spider being developed.

We'll be using the rvest package to analyze the html content, feel free to use any one.

We start by loading the required libraries and defining two global variables:

```
#!/usr/bin/env Rscript
suppressMessages(library(scrapystreaming))
suppressMessages(library(rvest))

pending_requests <- 0
result <- list()
```

It's important to use the `suppressMessages` command to avoid unecessary data in the process stdout.

Then, we define two functions:

- **response_parse** - parse the initial page, and then open a new request to each subcategory
- **response_category** - parse the subcategory page, getting the links and saving it to the `result` variable.

```
# function to get the initial page
response_parse <- function(response) {
    html <- read_html(response$body)

    for (a in html_nodes(html, "#subcategories-div > section > div > div.cat-item > a
↪")) {
        # we count the number of requests using this var
```

```
        pending_requests <<- pending_requests + 1
        # open a new request to each subcategories
        send_request(sprintf("http://www.dmoz.org%s", html_attr(a, "href")), response_
↪category)
    }
}

response_category <- function(response) {
    # this response is no longer pending
    pending_requests <<- pending_requests - 1

    html <- read_html(response$body)
    # get div with link and title
    for (div in html_nodes(html, "div.title-and-desc a")) {
        result[html_text(div, trim = TRUE)] <<- html_attr(div, "href")
    }

    # if finished all requests, we can close the spider
    if (pending_requests == 0) {
        f <- file("outputs/dmoz_data.json")
        # serialize the extracted data and close the spider
        write(toJSON(result), f)
        close(f)
        close_spider()
    }
}
```

Notice that when using the *send_request()*, we pass the `parse_category` function as the callback. Therefore, each response coming from this request will execute the `parse_category` function.

Finally, we start and run the spider, using:

```
create_spider("dmoz", "http://www.dmoz.org/Computers/Programming/Languages/Python/",␣
↪response_parse)
run_spider()
```

then, just save your spider and execute it using:

```
scrapy streaming name_of_script.R
```

or:

```
scrapy streaming Rscript -a name_of_script.R
```

1.8 Node.js Package

Todo

publish it and add the link to npm package with the package docs here

We provide a helper library to help the development process of external spiders in JavaScript using Node.js.

It's recommended to read the *Quickstart - Dmoz Streaming Spider* before using this package.

1.8.1 Installation

To install the `scrapystreaming` package, runs:

```
npm install scrapystreaming
```

and loads it using:

```
var scrapy = require('scrapystreaming');
```

1.8.2 scrapystreaming

The `scrapystreaming` Node package provide the following commands:

- `createSpider()`
- *`closeSpider()`*
- *`sendLog()`*
- *`sendRequest()`*
- *`sendFromResponseRequest()`*
- *`runSpider()`*

Tip: The Scrapy Streaming and your spider communicates using the system `stdin`, `stdout`, and `stderr`. So, don't write any data that is not a json message to the system `stdout` or `stderr`.

These commands write and read data from `stdin`, `stdout`, and `stderr` when necessary, so you don't need to handle the communication channel manually.

`create_spider` (*name*, *startUrls*, *callback*[, *allowedDomains*, *customSettings*])

Parameters

- **`name`** (*`string`*) – name of the
- **`startUrls`** (*`array`*) – list of initial
- **`callback`** (*`Function`*) – callback to handle the responses from
- **`allowedDomains`** (*`array`*) – list of allowed
- **`customSettings`** (*`object`*) – custom settings to be used in Scrapy

This command is used to create and run a Spider, sending the *`spider`* message.

Usages:

```
var callback = function(response) {
    // handle the response message
};

// usage 1, all parameters
scrapy.createSpider('sample', ['http://example.com'], callback,
                    ['example.com'], {some_setting: 'some value'});
// usage 2, empty spider

scrapy.createSpiders("sample", [], parse);
```

closeSpider()

Closes the spider, sending the *close* message.

Usage:

```
scrapy.closeSpider();
```

sendLog(*message*, *level*)

Parameters

- **message** (*string*) – log message
- **level** (*string*) – log level, must be one of 'CRITICAL', 'ERROR', 'WARNING', 'INFO', and 'DEBUG'

Send a log message to the Scrapy Streaming's logger output. This commands sends the `log` message.

Usages:

```
// logging some error
sendLog("something wrong", "error")
```

sendRequest(*url*, *callback*, *config*)

Parameters

- **url** (*string*) – request url
- **callback** (*function*) – response callback
- **config** (*object*) – object with extra request parameters (optional)
- **config.base64** (*boolean*) – if true, converts the response body to base64. (optional) :param string config.method request method (optional)
- **config.meta** (*object*) – request extra data (optional)
- **config.body** (*string*) – request body (optional)
- **config.headers** (*object*) – request headers (optional)
- **config.cookies** (*object*) – rqeuest extra cookies (optional)
- **config.encoding** (*string*) – default encoding (optional)
- **config.priority** (*int*) – request priority (optional)
- **config.dont_filter** (*boolean*) – if true, the request don't pass on the request duplicate filter (optional)

Open a new request, using the *request* message.

Usages:

```
var callback = function(response) {
    // parse the response
};

scrapy.sendRequest('http://example.com', callback);

// base64 encoding, used to request binary content, such as files
var config = {base64: true};
scrapy.sendRequest('http://example.com/some_file.xyz', callback, config)
```

sendFromResponseRequest(*url*, *callback*, *fromResponseRequest*, *config*)

Parameters

- **url** (*string*) – request url
- **callback** (*Function*) – response callback
- **fromResponseRequest** (*object*) – Creates a new request using the response
- **fromResponseRequest.base64** (*boolean*) – if true, converts the response body to base64. (optional)
- **fromResponseRequest.method** (*string*) – request method (optional)
- **fromResponseRequest.meta** (*object*) – request extra data (optional)
- **fromResponseRequest.body** (*string*) – request body (optional)
- **fromResponseRequest.headers** (*object*) – request headers (optional)
- **fromResponseRequest.cookies** (*object*) – rqeuest extra cookies (optional)
- **fromResponseRequest.encoding** (*string*) – default encoding (optional)
- **fromResponseRequest.priority** (*int*) – request priority (optional)
- **fromResponseRequest.dont_filter** (*boolean*) – if true, the request don't pass on the request duplicate filter (optional)
- **fromResponseRequest.formname** (*string*) – FormRequest.formname parameter (optional)
- **fromResponseRequest.formxpath** (*string*) – FormRequest.formxpath parameter (optional)
- **fromResponseRequest.formcss** (*string*) – FormRequest.formcss parameter (optional)
- **fromResponseRequest.formnumber** (*int*) – FormRequest.formnumber parameter (optional)
- **fromResponseRequest.formdata** (*object*) – FormRequest.formdata parameter (optional)
- **fromResponseRequest.clickdata** (*object*) – FormRequest.clickdata parameter (optional)
- **fromResponseRequest.dont_click** (*boolean*) – FormRequest.dont_click parameter (optional)
- **config** (*object*) – object with extra request parameters (optional)
- **config.base64** (*boolean*) – if true, converts the response body to base64. (optional)
- **config.method** (*string*) – request method (optional)
- **config.meta** (*object*) – request extra data (optional)
- **config.body** (*string*) – request body (optional)
- **config.headers** (*object*) – request headers (optional)
- **config.cookies** (*object*) – request extra cookies (optional)
- **config.encoding** (*string*) – default encoding (optional)
- **config.priority** (*int*) – request priority (optional)

- **config.dont_filter** (*boolean*) – if true, the request don't pass on the request duplicate filter (optional)

This function creates a request, and then use its response to open a new request using the *from_response_request* message.

Usages:

```
var callback = function(response) {
    // parse the response
};

// submit a login form, first requesting the login page, and then submitting the form

// we first create the form data to be sent
var fromResponseRequest = {
    formcss: '#login_form',
    formdata: {user: 'admin', pass: '1'}
};

// and open the request
scrapy.sendFromResponseRequest('http://example.com/login', callback,␣
↪sendFromResponseRequest);
```

runSpider([*exceptionHandler*])

Parameters **exceptionHandler** (*function*) – function to handle exceptions. Must receive a single parameter, the received json with the exception. (optional)

Starts the spider execution. This will bind the process stdin to read data from Scrapy Streaming, and process each message received.

If you want to handle the exceptions generated by Scrapy, pass a function that receives a single parameter as an argument.

By default, any exception will stop the spider execution and throw an Error.

Usage:

```
// create the spider

...
scrapy.createSpiders("sample", ['http://example.com'], parse);

// and start to listen the process stdin
scrapy.runSpider();

// with exception listener
scrapy.runSpider(function(error){
    // ignores the exception
});
```

1.8.3 Dmoz Streaming Spider with R

In this section, we'll implement the same spider developed in *Quickstart - Dmoz Streaming Spider* using the `scrapystreaming` package. It's recommended that you have read the quickstart section before following this topic, to get more details about Scrapy Streaming and the spider being developed.

We'll be using the cheerio package to analyze the html content, feel free to use any one.

We start by loading the required libraries and defining two global variables:

```
#!/usr/bin/env node

var scrapy = require('scrapystreaming');
var jsonfile = require('jsonfile');
var cheerio = require('cheerio');

var pendingRequests = 0;
var result = {};
```

Then, we define two functions:

- **parse** - parse the initial page, and then open a new request to each subcategory
- **parse_cat** - parse the subcategory page, getting the links and saving it to the `result` variable.

```
// function to parse the response from the startUrls
var parse = function(response) {
    // loads the html page
    var $ = cheerio.load(response.body);

    // extract subcategories
    $('#subcategories-div > section > div > div.cat-item > a').each(function(i, item)
↪{
        scrapy.sendRequest('http://www.dmoz.org' + $(this).attr('href'), parse_cat);
        pendingRequests++;
    });
};

// parse the response from subcategories
var parse_cat = function(response) {
    var $ = cheerio.load(response.body);

    // extract results
    $('div.title-and-desc a').each(function(i, item) {
        result[$(this).text().trim()] = $(this).attr('href');
    });

    pendingRequests--;
    // if there is no peding requests, save the result and close the spider
    if (pendingRequests == 0) {
        jsonfile.writeFile('outputs/dmoz_data.json', result);
        scrapy.closeSpider();
    }
};
```

Notice that when using the *sendRequest()*, we pass the `parse_cat` function as the callback. Therefore, each response coming from this request will execute the `parse_cat` function.

Finally, we start and run the spider, using:

```
scrapy.createSpider('dmoz', ["http://www.dmoz.org/Computers/Programming/Languages/
↪Python/"], parse);
scrapy.runSpider();
```

then, just save your spider and execute it using:

```
scrapy streaming name_of_script.js
```

or:

```
scrapy streaming node -a name_of_script.js
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

P

R

S

T

W