ScrapQD

Release v1.0

Durai Pandian

Mar 15, 2022

ALL CONTENTS

Table of Contents

CHAPTER

ONE

INTRODUCTION

ScrapQD consists of query definition created for scraping web data using GraphQL-Core which is port of GraphQL.js.

Library intends to focus on how to locate data from website and eliminate backend process of crawling. So people can just have xpath and get data right away.

It supports scraping using requests for traditional websites and selenium for modern websites (js rendering). Under selenium it supports Google Chrome and FireFox drivers.

ScrapQD library only uses lxml parser and xpath used to locate elements.

CHAPTER

TWO

GETTING STARTED

How to install

```
pip install scrapqd
```

How to run the server standalone

You can run scrapqd graphql server standalone without any additional code with below command. Flask is used as server and localhost.

```
python -m scrapqd
```

Flask uses 5000 as default port. You can change the port and host with below options.

```
python -m scrapqd --port 5001 --host x.x.x.x
```

CHAPTER

THREE

SCRAPQD

ScrapeQD consists of below components.

- *GraphQL UI*
- Query
  - Query Type -> Document (consists of leaf type and group type)
  - Group Type
  - Leaf Type
- Parser
- Executor

CHAPTER

FOUR

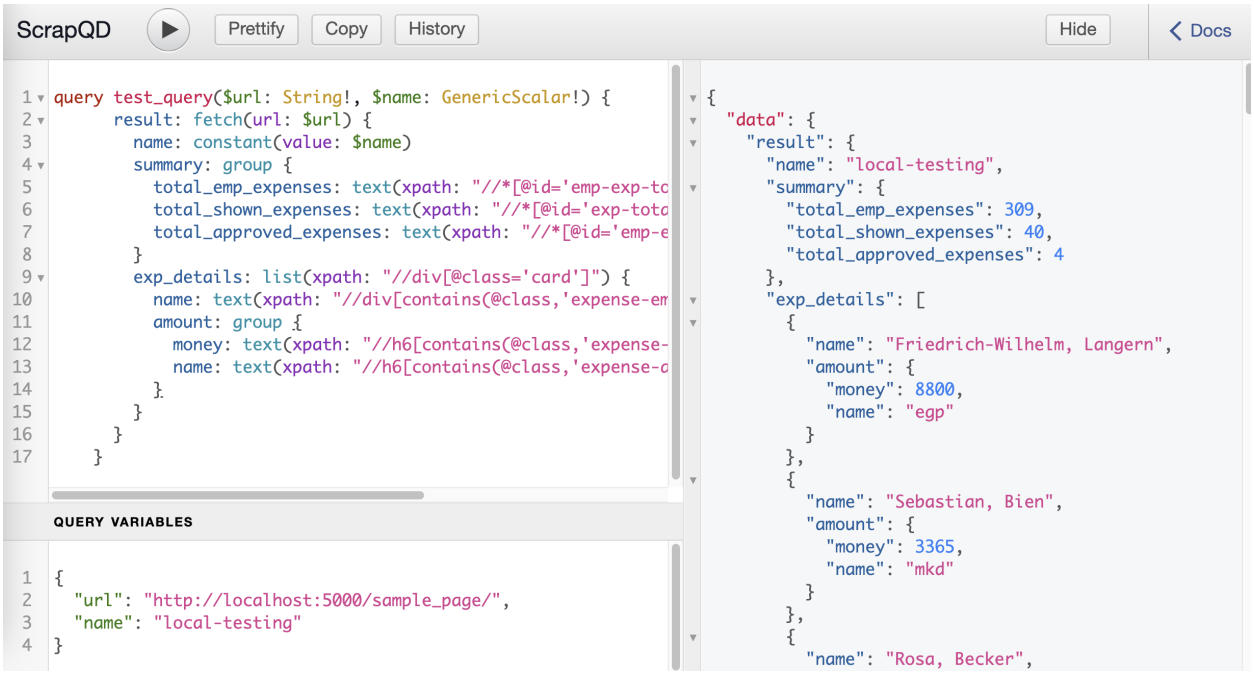
GRAPHQL UI

GraphQL UI provides flexibility to write scrapqd query and test it. GraphQL UI supports auto completion and query documentation to develop query faster. You can access UI - localhost.

UI is loaded with sample query and Sample page is accessible for practice.

You can pass custom template for the query ui.

- History - to view past 10 queries that was tested.
- Copy - Copy the content in the query window.
- Prettify - prettifies the graphql query.
- Show/Hide - Show or hides the result window.
- Query Variables - query variables editor to pass data to query when you execute.



CHAPTER

FIVE

EXECUTING WITH CLIENT

```
from scrapqd.client import execute_sync

query = r"""
        query test_query($url: String!, $name: GenericScalar!) {
          result: fetch(url: $url) {
            name: constant(value: $name)
            summary: group {
              total_shown_expenses: regex(xpath: "//*[@id='exp-total']", pattern: "(\\d+)
↪")
            }
          }
        }"""

query_variables = {
    "url": "http://localhost:5000/scrapqd/sample_page/",
    "name": "local-testing"
}
result = execute_sync(self.query, query_variables)
```

CHAPTER

SIX

INTEGRATING WITH EXISTING FLASK APP

6.1 Sample Flask app

```
from flask import Flask

name = __name__
app = Flask(name)

@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"
```

6.2 Integrating scrapqd with existing app

```
from scrapqd.app import register_scrapqd
register_scrapqd(app,
                 register_sample_url=True,
                 redirect_root=True)
```

app: Flask application

register_sample_url: `False` will not register sample page url to Flask application. Default is `True`

redirect_root: Redirect root url to graphql ui if this is set to `True`. This will not reflect, if there is already root route defined as above example.

CHAPTER

SEVEN

TEST (FOR DEVELOPMENT)

- Clone the github repository

```
git clone https://github.com/dduraipandian/scrapqd.git
```

- create virtual environment to work

```
pip3 install virtualenv
virtualenv scrapqd_venv
source scrapqd_venv/bin/activate
```

- install tox

```
pip install tox
```

- run tox from the project root directory
  - current tox have four python version - py37,py38,py39,py310
  - check your python version

```
python3 --version

# Python 3.9.10
```

  - once you get your version (example: use py39 for 3.9) to run tox

```
tox -e py39
```

CHAPTER

EIGHT

FAQS

- How to copy query from graphql ui to python code.
  - you can normally copy code from ui to python code to execute using client.
  - if you hav `regex` query, patterns needs to escaped in the python code. In such, use python raw strings, where backslashes are treated as literal characters, as above example.
- How to suppress webdriver logs
  - If you see webdriver logs like below, set `WDM_LOG_LEVEL=0` as environment variable and run

```
[INFO] [97002] [2022-03-14T02:18:26+0530] [SCRAPQD] [/webdriver_manager/
↪logger.py:log():26] [WDM] [Driver [/99.0.4844.51/chromedriver] ...]
```

- How to change log level for scrapqd library
  - `ERROR` level is default logging. You can change this with `SCRAPQD_LOG_LEVEL` environment variable.

8.1 Query

Scrape Query can be created with query, group and leaf queries.

- ***Query Type***
  - *fetch*
  - *selenium*
- ***Group Type***
  - *group*
  - *list*
- ***Leaf Type***
  - *constant*
  - *text*
  - *attr*
  - *link*
  - *query_params*
  - *form_input*
  - *regex*

8.1.1 Sample Query

```
query test_query($url: String!, $name: GenericScalar!) {
  result: fetch(url: $url) {
    name: constant(value: $name)
    summary: group {
      total_emp_expenses: text(xpath: "//*[@id='emp-exp-total']", data_type: INT)
      total_shown_expenses: text(xpath: "//*[@id='exp-total']/span[2]", data_type: INT)
      total_approved_expenses: text(xpath: "//*[@id='emp-exp-approved']/span[2]", data_
↪type: INT)
    }
    exp_details: list(xpath: "//div[@class='card']") {
      name: text(xpath: "//div[contains(@class,'expense-emp-name')]")
      amount: group {
        money: text(xpath: "//h6[contains(@class,'expense-amount')]/span[1]", data_type:␣
↪INT)
        name: text(xpath: "//h6[contains(@class,'expense-amount')]/span[2]")
      }
    }
  }
}
```

Query variables

```
{
    "url": "http://localhost:5000/scrapqd/sample_page/",
    "name": "local-testing"
}
```

8.1.2 Query Type

Query type queries are used for crawling url using different executors and pass down the data to child queries Leaf type for further processing. They expect leaf queries as sub query.

fetch

fetch(*url*, *headers={}*, *executor='requests'*, *is_json_response=false*, *method='GET'*, *cache=false*)

Fetch query will crawl the traditional websites.

url

URL to crawl

headers

- sometimes websites need additional headers in the request. By default, system provides below headers. The given headers will be updated with default headers. So default system headers are constant which will be sent for all the request.
  - User-Agent: from the data files. This can be changed using USER_AGET_DATA_FILE or USER_AGET_DATA config.
  - Connection: keep-alive
  - Upgrade-Insecure-Requests: 1

  - Accept-Language: en-US,en;q=0.9
  - Accept-Encoding: gzip, deflate, br
  - Pragma: no-cache
- You might not need this for most website. API type urls might need other extra headers and other http methods.

executor

- Executors define how to crawl the url and how to process their response. By default system has "requests" executors which supports Requests library.
- Custom executors can be creating by extending Executor class.

is_json_response

- It is by default False. You have to set True if the url returns json data. Processing of json data is not supported as of now. This is for future enhancement. System will throw error if this is set to True.

method

- http method to use for the request.
- System uses **GET** by default. For website crawl you do not need to set this parameter.
- API type urls might need other http methods like **POST**.

cache

Note: This should be used in development period

- Fetch will be time consuming as it gets website data from internet. While developing the query, you may run the query multiple times. It will affect the development time.
- Setting `cache = true` will cache the result of the url for consequent same url.
- Setting `ENV=DEVELOPMENT` in config will enable cache for all the queries by default. Anything other than development, cache parameter is ignored.

selenium

selenium(*url*, *browser=GOOGLE_CHROME*, *options={}*, *is_json_response=false*, *cache=false*)

Selenium query will crawl the modern websites with javascript rendering.

url
URL to crawl.

browser
System supports below browser.

- GOOGLE_CHROME
- FIREFOX

options
Additional options to be used in crawling using selenium.

- `xpath` Selenium will wait this element to be present in the loaded webpage.
- `wait_time` Selenium will wait for above xpath target (wait_time) secs.

is_json_response
It is by default False. You have to set True if the url returns json data. Processing of json data is not supported as of now. This is for future enhancement. System will throw error if this is set to True.

cache
Similar to cache parameter in fetch query.

8.1.3 Group Type

Group queries process groups multiple leaf nodes and process multiple results of a xpath. They expect leaf or group queries as sub query.

- group
- list

group

Group query will group the leaf node output under group variable to returns result to client. This will be helpful to group certain types of elements/data from the query without needing addition outside code.

```
amount: group {
    money: text(xpath: "//h6[contains(@class,'expense-amount')]/span[1]", data_type: INT)
    name: text(xpath: "//h6[contains(@class,'expense-amount')]/span[2]")
}
```

list

list(*xpath*)
List query will help you to write sub-query to extract data from the parent and returns. If the list xpath return multiple elements, sub-query applied on each item in the list.

xpath
to locate element

Example

```
exp_details: list(xpath: "//div[@class='card']") {
    name: text(xpath: "//div[contains(@class,'expense-emp-name')]")
    amount: group {
        money: text(xpath: "//h6[contains(@class,'expense-amount')]/span[1]", data_type:␣
↪INT)
        name: text(xpath: "//h6[contains(@class,'expense-amount')]/span[2]")
    }
}
```

Result

```
{
    "result": {
        "exp_details": [
            {
              "name": "Friedrich-Wilhelm, Langern",
              "amount": {
                "money": 8800,
                "name": "egp"
              }
            },
            {
              "name": "Sebastian, Bien",
              "amount": {
                "money": 3365,
                "name": "mkd"
              }
            },
            {
              "name": "Rosa, Becker",
              "amount": {
                "money": 6700,
                "name": "xof"
              }
            },
            {
              "name": "Ines, Gröttner",
              "amount": {
                "money": 8427,
                "name": "npr"
              }
            },
            {
              "name": "Clarissa, Bonbach",
              "amount": {
                "money": 1609,
                "name": "fjd"
              }
            },
            {
              "name": "Zbigniew, Stolze",
              "amount": {
                "money": 8789,
                "name": "ern"
              }
            },
            {
              "name": "Ines, Mentzel",
              "amount": {
                "money": 1750,
                "name": "srd"
              }
            }
          ],
```


(continues on next page)

(continued from previous page)

```
    }
}
```

8.1.4 Leaf Type

Leaf nodes are final queries to get the value from html element such as `text` from above query. You can not provide another leaf query as sub query.

- *constant*
- *text*
- *attr*
- *link*
- *query_params*
- *form_input*
- *regex*

Data Types

Few leaf queries support data types. If the data type is given, the element content will be converted to the given data type and sent to client. System supported below data types. Custom data types can be created as well.

TEXT
Default data type.

RAW
When the element text is extract, text might have extra whitespace. They are stripped away by default. When RAW data type is given, data will be sent as it is extracted from the element.

INT

- Data is converted to integer.
- Example
  - 1,024 -> 1024
  - 12K -> 12000 (k/K - thousand, m/M - million, b/B - billion)

FLOAT

- Data is converted to decimal.

Multi

Leaf nodes support multi parameter. Xpath will locate multiple elements. This parameter will help the system who to process and return to client.

- `false` Only first element will be processed and returned to the client.
- `true` All the elements will be processed. Result will be sent as array/list to client. If the query supports data_type parameter, data_type conversion will be applied on all elements.

When multi is set false, result format will be not same when it is set to true.

you can set NON_MULTI_RESULT_LIST to `True` to have same format on both cases in the config file.

constant

constant(*value*)

Constant query will give back results to client as hard coded in the query or value passed from query variables.

value

Non null value in the query or can be passed from query variable as from the example.

```
name: constant(value:"local-testing")
```

text

text(*xpath*, *data_type: TEXT*, *multi: false*)

Text query will get the content of the given element. Text does not represent that it will return text. It simply denotes that it will extract text from element.

xpath

Path to locate element

data_type

Data type to return

multi

when xpath matches multiple elements,

- `False` Processes first element
- `True` Processes all elements

Example

```
total_emp_expenses: text(xpath: "//*[@id='emp-exp-total']", data_type: INT)
```

attr

attr(*xpath*, *name=null*, *multi=false*)

Element will have multiple attributes as below. Attr query will help to fetch all of them or specified one. Data-hovercard-type, href are `attributes` on the example element. It will extract attributes value as key, value pair. Key as name, value as value of the attribute.

name

- If the name is not given, it will extract all the attributes.

- For example, if the name = 'href' given, it will get "{href: /abcxcom}" mapping.

multi
when xpath matches multiple elements,

- `False` Processes first element
- `True` Processes all elements

Example

```
approval_id: attr(xpath: "//button[contains(@class, 'expense-approve')]", name: "id")
```

link

link(*xpath*, *base_url=null*, *multi=false*)
In html, anchor <a> tag defines link to another web page. With link query, you can get entire url with ease. There are times websites use relative url.

Link query construct full url from the requested url automatically. You can override the parent url with base_url parameter in the query.

xpath
Path to locate element

base_url
Custom url to create absolute url

multi
when xpath matches multiple elements,

- `False` Processes first element
- `True` Processes all elements

Example

```
website : link(xpath:"//a[contains(@class, 'site-link')]")
```

query_params

query_params(*xpath*, *name: null*, *multi: false*)
When you want to extract query parameter from url in anchor tag or any element has url type content, you can use `query_params` query.

xpath
Path to locate element

name

- If the name is not given, it will extract all the query parameters in the url.
- For example, if the name = 'product' given, it will get "{product: xyzcourse}" mapping.

multi
when xpath matches multiple elements,

- `False` Processes first element

- `True` Processes all elements

Example

```
user_id: query_params(xpath:"//a/@href", name: "user")
```

Result

regex

regex(*xpath*, *pattern*, *source="TEXT"*, *multi: false*)

Regex will be used on the located element using xpath and returns the result.

xpath
Path to locate element

pattern
Regular expression pattern to match and it will be used in re.findall from python to extract data.

source
Regular expression can be applied on located element's content or element's source html itself.

- `text` Regex will be applied on element's content. This is default value.
- `html` Regex will be applied on element's html.

multi
when xpath matches multiple elements,

- `False` Processes first element
- `True` Processes all elements

Example

```
total_shown_expenses: regex(xpath: "//*[@id='exp-total']", pattern: "(\\d+)")
```

Result

```
"total_shown_expenses": [
    "40"
]
```

form_input

form_input(*xpath*, *name: null*, *multi: false*)

Form input query will help you to extract input elements name, value pair from form element.

xpath
Path to locate form element

name

- If the name is not given, it will extract all the input elements under the form.
- If the name is given, it will get input element with the given name.

multi
when xpath matches multiple elements,

- `False` Processes first element
- `True` Processes all elements

Example

Html

```
<form class="requestParams" id="apiAttr">
    <input name="rlz" value="1C5CHFA_enIN991IN991" type="hidden">
    <input name="tbm" value="lcl" type="hidden">
    <input name="sxsrf" value="APq-WBu3vzrA9-WQU_Mp0Zs9aq2a-PQlJg:1644327612221" type=
↪"hidden">
    <input value="vHICYpKHDaWXseMP57uWuA4" name="ei" type="hidden">
    <input value="AHkkrS4AAAAAYgKAzF3dfuu_a7YROtX7wSMb404M2sTE" disabled="true" name=
↪"iflsig" type="hidden">
</form>
```

Query

```
meta_data: form(xpath: "//form[@class='requestParams']", name: "sxsrf")
```

8.2 Query Fields

Query fields are GraphQL fields. scrapqd.gql has all the graphql related implementation. Refer GraphQL documentation for more information.

Query type, Leaves type and group are categorized in scrapqd based on their role. But for graphql all the fields are created in same manner.

8.2.1 How to create query fields

Query fields are creating using `GraphQLField` attaching to resolver function. Resolver function will be invoked by graphql to process the query.

Resolver function

resolver(*parser: Parser*, *info: ResolveInfo*, *xpath*, ***kwargs*)

Resolver function for the graphql field.

parser

Parser instance passed down from parent query.

info

GraphQLResolveInfo instance which gives resolver information.

xpath

path to locate node(tag).

kwargs

any additional parameters defined in the GraphQL field.

GraphQL Field

class Field

GraphQL field class is used to create scrapqd query

type
GraphQL field type. Mostly `GenericScalar` type is used for fields in the scrapqd library.

args
Dictionary of arguments for the field in query ex: xpath, name. This should be argumented in resolver function above.

resolve
resolver function which will be invoked while querying. Above resolver function should be given here.

description
This description will be shown in the graphql ui for documentation.

8.2.2 Example: Text field

Resolver function

```
@with_error_traceback
def resolve_text(parser: Parser, info: ResolveInfo,
                 xpath, data_type=const.DATA_TYPE_DEFAULT_VALUE, multi=const.MULTI_
↪DEFAULT_VALUE):
    """Extracts node(tag) content using given XPath.

    :param parser: Parser instance passed down from parent query.
    :param info: GraphQLResolveInfo instance which gives resolver information.
    :param xpath: path to locate node(tag).
    :param data_type:   Extracted text will be always in text format. When the data type␣
↪is provided,
                        content is converted to that format and returned to the client.
                        Accepted data types:

                            - text (default)
                            - int
                            - float

    :param multi: by default, it is set to False. Thus, when the given xpath locates␣
↪multiple nodes,
                it returns first node value. if it is set `true`, it will return all the␣
↪node values" \
                as list.Given data type is applied to all the nodes individually.
    :return:

            - text - when multi is set to False, This option can be overridden to return␣
↪list with single value using `NON_MULTI_RESULT_LIST`.
            - List - when multi is set to True
    """
    key = get_key(info)
    parser.datatype_check(key, data_type)
    result = parser.extract_text(key=key, multi=multi, xpath=xpath)
    result = parser.data_conversion(result, data_type)
    result = parser.get_multi_results(multi, result)
    parser.caching(key, result)
    return result
```

Query Field

```
text = Field(GenericScalar,
        args={
            'xpath': Argument(NonNull(String), description=const.xpath_desc),
            'data_type': Argument(DataTypeEnum, description="data type which should be␣
↪converted"),
            'multi': Argument(Boolean, description=const.multi_desc),
        },
        resolve=resolve_text,
        description="Extracts text content from the give xpath")
```

8.3 Executor

Executor is a crawler engine to scrape data. Any custom executor extends Executor interface and implements abstract method.

- *Executor Interface*
- *Requests*
- ***Selenium***
  - *Selenium Driver*
  - *Selenium Browser*
  - *Selenium Executor*

8.3.1 Executor Interface

Understanding executor interface is crucial to understand default executors and creating custom executors.

class scrapqd.fetch.interface.**Executor**(*url*, *method='get'*, *headers=None*, *response_type=None*)
Interface for Executor implementation

This class is exported only to assist people in implementing their own executors for crawling without duplicating too much code.

property success_status_code
Default success code for the request. Default success codes are [200].

Returns List

get_payload(*payload*)
Creates payload for http request.

Parameters **payload** – Additional payload argument for request.

Returns Dict

get_default_headers()
Get user-agent and constructs other default headers for the request.

- User-Agent: from the data files.
- Connection: keep-alive
- Upgrade-Insecure-Requests: 1

- Accept-Language: en-US,en;q=0.9
- Accept-Encoding: gzip, deflate, br
- Pragma: no-cache

Returns Dict

`get_response_type()`
Gets response type from the request response.

Returns String

`get_headers()`
Constructs headers to be applied to the request from default headers and user provided headers. User provided headers will override default headers.

Returns Dict

`get_response_content()`
gets response content from processed request.

Returns

- `json` If the response type is json
- `html` If the response type is text/html

`execute(**kwargs)`
Executes crawl method and gets http response from web.

Parameters `kwargs` – Additional keyword arguments for extensibility.

Raises `Exception` – Re-raises the exception occurred in the block for client to capture and handle

`abstract get_response_url()`
Gets response url. It should be the final url after redirect (if any).

Returns String

`abstract get_response_headers()`
Gets http response headers

Returns Dict

`abstract is_success()`
Method definition to identify the request is successful or not. By default, status_code == 200 is considered as success.

Returns Boolean

`abstract get_response_text()`
Gets response text.

Returns String

`abstract get_response_json()`
Gets response as json.

Returns Dict

`abstract get_status_code()`
Gets response status code of the http request made.

Returns integer

abstract crawl(*url*, *method='get'*, *headers=None*, ***kwargs*)
Crawls given url from web. This method should return only http response from the library without any further processing of the response.

Parameters

- **url** – URL to crawl
- **method** – Http method which should be used to crawl
- **headers** – Additional headers for executor. Some websites need addition headers to make request. System add below request headers by default. These headers can be overridden using header argument.
  - User-Agent: from the data files.
  - Connection: keep-alive
  - Upgrade-Insecure-Requests: 1
  - Accept-Language: en-US,en;q=0.9
  - Accept-Encoding: gzip, deflate, br
  - Pragma: no-cache
- **kwargs** – Additional keyword arguments to support executor.

Returns Http response

8.3.2 Requests

Requests uses requests library for executing requests and implements parent abstract methods.

```
class Requests(Executor):
    def get_response_url(self):
        return self.response.url

    def get_response_headers(self):
        return dict(self.response.headers)

    def get_status_code(self):
        return self.response.status_code

    def get_response_text(self):
        return self.response.content

    def get_response_json(self):
        return self.response.json()

    def is_success(self):
        status_code = self.get_status_code()
        return status_code in self.success_status_code

    def crawl(self, url, headers=None, method="get", **kwargs):
        return requests.request(self.method, self.url, headers=headers, **kwargs)
```

8.3.3 Selenium

Selenium Driver

SeleniumDriver is the generic implementation for crawling using selenium.

class scrapqd.executor.selenium_driver.selenium.**SeleniumDriver**

Internal selenium driver implementation for all the browser types

wait_load(*xpath*, *wait_time*)

Waits for browser to load specific element in the given url. If the xpath is not given, selenium will wait for the document to be ready.

Parameters

- **xpath** – Element to wait
- **wait_time** – Wait time in seconds for the element to present in the web page.

fetch(*url*, ***kwargs*)

Fetches web page for the url

Parameters

- **url** – url to crawl
- **kwargs** –
  - `wait` Wait time in seconds for the element in the web page.
  - `xpath` Element to wait. If this parameter is not given, selenium will wait for the document

  to be ready till wait time.

get_response_headers()

This executes javascript in the browser to get http response headers.

Returns Dict

get_current_url()

Gets the current url after redirect (if any).

Returns String

get_page_source(*url*, ***kwargs*)

Returns page source of the url

Parameters

- **url** – url to crawl
- **kwargs** –
  - `wait` Wait time in seconds for the element in the web page.
  - `xpath` Element to wait. If this parameter is not given, selenium will wait for the

  document to be ready till wait time.

Returns HTML Web page string

clean_up()

Quits browser and sets None, when this method is called

classmethod get_executable_path(*browser*, ***kwargs*)

Gets browser executable from repository using *webdriver_manager*.

Parameters

- **browser** – Name of the browser
- **kwargs** – Webdriver_manager options for the browser to download executable.

Returns BrowserDriver

Selenium Browser

GoogleChrome, Firefox browsers are implemented currently. GoogleChrome is given as example here.

class scrapqd.executor.selenium_driver.browsers.**GoogleChrome**
Creates Google Chrome type driver

classmethod create_browser()
Returns headless Google browser object

Selenium Executor

Selenium executor is used to crawl modern webpages which uses javascript rendering (client-side rendering).

```
class Selenium(Executor):
    """SeleniumExecutor is class a generic processor (facade) for all browsers and
    implements all abstract method from `Executor` class."""

    def __init__(self, url, **kwargs):
        super().__init__(url, **kwargs)
        self._response_headers = {}
        self._current_url = None

    def get_response_url(self):
        if not self._current_url:
            logger.error("Not able to get current_url for %s from selenium", self.url,␣
↪exc_info=True)
            return self.url
        return self._current_url

    def is_success(self):
        return True

    def get_response_text(self):
        return self.response

    def get_response_json(self):
        if isinstance(self.response, str):
            try:
                self.response = json.loads(self.response)
            except Exception:
                logger.exception("Not able to get convert to json data %s", self.url,␣
↪exc_info=True)

        return self.response
```

(continues on next page)

(continued from previous page)

```
    def get_status_code(self):
        return 200

    def get_response_headers(self):
        return self._response_headers

    def crawl(self, url, method="get", headers=None, **kwargs):
        """Selenium crawl gets browser from browser factory and crawls the url"""
        browser_name = kwargs.get("browser", "GOOGLE_CHROME")
        browser = BrowserFactory().get(browser_name)()
        response = browser.get_page_source(url, **kwargs)
        self._response_headers = browser.get_response_headers()
        self._current_url = browser.get_current_url()
        return response
```

8.4 Parser

Parser is used in the GraphQL query to parse the html. Current system supports xpath in Lxml parser.

Library does not support Beautiful soup as it slower than lxml parser and Selector parsing is comparatively slower than xpath.

- *Lxml*

8.4.1 Lxml

class scrapqd.gql_parser.lxml_parser.**LXMLParser**(*raw_html=None*, *html_tree=None*)

This is concerete implementation for lxml gql_parser to parse html text.

xpath_element(*element*, *xpath=None*, ***kwargs*)

Extracts target node using xpath from given html element.

Parameters

- **element** – Html element.
- **xpath** – Xpath to locate the elements.
- **kwargs** – Additional keyword arguments for extensibility.

Returns List[HTMLElement]

xpath_text(*element*, *xpath*, ***kwargs*)

Extracts text for given xpath.

Parameters

- **element** – Html element.
- **xpath** – Xpath to locate the elements.
- **kwargs** – Additional keyword arguments for extensibility.

Returns List[String]

extract_element_source_text(*element*)

Extracts source html content

Parameters **element** – Html element.

Returns String

extract_text(*xpath*, ***kwargs*)
Extracts text content from element.

Parameters

- **xpath** – Xpath to locate the elements.
- **kwargs** – Additional keyword arguments for extensibility.

Returns List[String]

extract_elements(*xpath*, ***kwargs*)
Extracts nodes from given html element.

Parameters

- **xpath** – Xpath to locate the elements.
- **kwargs** – Additional keyword arguments for extensibility.

Returns List[HTMLElement]

extract_attr(*xpath*, ***kwargs*)
Extracts attributes from the html element.

Parameters

- **xpath** – Xpath to locate the elements.
- **kwargs** – Additional keyword arguments for extensibility.

Returns List[Dict]

extract_form_input(*xpath*, ***kwargs*)
Extracts form inputs using given xpath. Method expects xpath to locate form node.

Parameters

- **xpath** – Xpath to locate the elements.
- **kwargs** – Additional keyword arguments for extensibility.

Returns List[Dict]

8.5 Settings

ScrapQD uses below default configuration to function properly. Below configs can be overridden by user config. Default config is located here.

- *Config*
  - *APP_NAME*
  - *CRAWLERS*
  - *LEAVES*
  - *QUERY_FIELDS*
  - *BROWSERS*
  - *DATATYPE_CONVERSION*

8.5.1 Config

APP_NAME

Default app name is ScrapQD. You can change this from config.

CRAWLERS

Requests and Selenium are system crawlers. If the defines customer executor, it needs to defined in the config.

LEAVES

Application owner can define custom leaves for their use case and provide in config to use in the query. Leaves are explained here.

QUERY_FIELDS

Application owner can define additional query fields (ex: puppeteer) and provide in the config. Queries are explained here.

BROWSERS

System uses GOOGLE_CHROME, FIREFOX browser in selenium to crawl modern webpages (javascript rendering). Custom browser can be created using Browser class and update this configuration.

DATATYPE_CONVERSION

Additional custom data type conversion mapping for the application.

NON_MULTI_RESULT_LIST

Config whether to send result as List or return single element when multi=False in the leaf nodes. You can read more from here.

LOCAL_CACHE_TTL

Fetch results are cached in local memory to speed up development. However, lifetime of the cache will be 10 minutes by default. You can update this config to change ttl of cache. You can read more about this here.

USER_AGET_DATA_FILE

User-Agent is added to each request headers while using fetch query. ScrapQD library has set of latest user agents in the file to load.

You can override them if you have your own user-agent files. Each user agent entry should on new line.

Note: Best to keep user-agent updated with latest agents on regular basis. Sites might return different format for older user agents.

USER_AGET_DATA

You can set this as list of user agents. System will use this and ignore *USER_AGET_DATA_FILE* config.

CHROMIUM_VERSION

System downloads latest version on chromium engine. You can set this to use the same version. Latest version will be downloaded by default.

GECKODRIVER_VERSION

If you are using firefox browser, you can set this to use specific gecko driver version. Otherwise latest version will be used.

DEFAULT_BROWSER

GOOGLE_CHROME is the default browser used in the library. You can update this to change to "FIREFOX".

8.5.2 How to Create config

You can add additional *LEAVES*, *CRAWLERS*, *QUERY_FIELDS* and *DATATYPE_CONVERSION*. But you can not override the system config.

You can override rest of the configs. You can create the file with configs as here and set *SCRAPQD_CONFIG* environment variable.

Example:

Your project files are under google_search and you have your config /google_search/configuration/scrapqd_config.py.

Your environment variable should be

```
SCRAPQD_CONFIG=configuration.scrapqd_config
```

8.6 How to Guide

Contents

8.6.1 How to add custom executors to system

- Understand Executor Interface
- Create your custom executor similar to Requests or Selenium.
- Add to the config - CRAWLERS. Example: Puppeteer

```
from crawler.executors import Puppeteer, SeleniumOther

CRAWLERS = {
    "PUPPETEER": Puppeteer,
    "SELENIUM_OTHER": SeleniumOther
}
```

- Override config
- Restart your application
- You should be able to use *select_options* as leaf query from the graphql ui.

8.6.2 How to add new leaves to system

- Understand Query fields
- Create your custom field similar to Example text field.
- Add to the config - LEAVES. Example: select_options. It should be dict(name: field object).

```
from crawlers.fields import select_options

LEAVES = {
    'select_options': select_options
}
```

- Override config

- Restart your application
- You should be able to use *select_options* as leaf query from the graphql ui.

8.6.3 How to add additional data type

- Understand Data Type
- Create your data type conversion function.

  Function should accept one value to process and return one value after conversion. Example function to boolean data conversion.

```
def boolean(value):
    if isinstance(value, int) or isinstance(value, float):
        value = False if value == 0 else True
    elif isinstance(value, bool):
        pass
    elif isinstance(value, str):
        if value.isdigit():
            value = False if float(value) == 0 else True
        else:
            try:
                value = float(value)
                value = False if value == 0 else True
            except:
                value = False if value == 'false' else True
    elif value is not None:
        value = True
    else:
        value = False
    return value
```

- Add to the config - DATATYPE_CONVERSION. Example: boolean. It should be dict(name: function).

```
from crawlers.data_types import boolean

LEAVES = {
    'boolean': boolean
}
```

- Override config
- Restart your application
- You should be able to use *boolean* as data type in the query.

8.6.4 How to add browsers to system

- Understand Browser implementation.
- Create your custom browser similar to GoogleChrome.
- Add to the config - BROWSER. Example: chromium. It should be dict(name: field object).

```
from crawlers.browsers import chromium

LEAVES = {
    'CHROMIUM': chromium
}
```

- Override config
- Restart your application
- You should be able to use *CHROMIUM* in the browser with selenium query.

INDEX