Writing Web Crawlers in Scrappy

By Juan Álvarez Martínez

As the Internet becomes more popular and widespread, more content is generated and shared on the Web. Anyone with the expertise or resources to exploit this trove of information will require a tool able to gather and interpret the bits of information from this massive content that are more relevant for a given application. Such a tool is called a **Web crawler** and for the most basic scenarios it's fairly easy to implement: start with a seed of URLs in a priority queue, fetch the pages from those locations, parse the content including any hyperlink the page may include and keep exploring these new links until a stop condition is met. In practice, however, there are challenges that arise when crawling multiple domains (broad crawling) or even a complex enough website. For example, parsing media in different formats other than XHTML, slow or overloaded servers that will struggle to cope with the crawling requests, bot detection mechanisms against crawlers, spam or redundant pages, freshness of content for pages that update frequently, servers that ban automated clients such as crawlers and managing the computing resources (network, CPU, memory and storage) required to crawl and parse massive amounts of web pages. In this tech review paper, I will describe a popular Web crawler framework named Scrapy (https://scrapy.org) and list some of the features and extensions that are available to address the challenges mentioned above. Scrapy is an open-source framework with a large community of developers and it's the preferred option for writing new crawling applications because of its popularity and small learning curve.

Scrapy's architecture

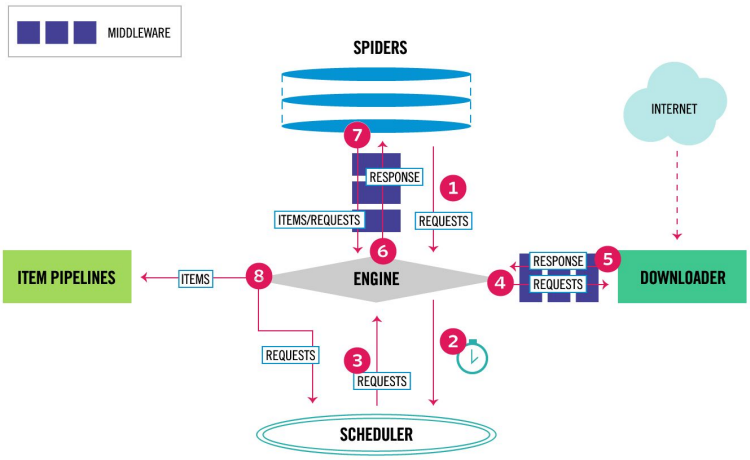

Fig 1. Scrapy's architecture from the documentation.
https://docs.scrapy.org/en/latest/topics/architecture.html. Accessed November 15, 2020.

As shown in the diagram above, Scrapy solves the Web Crawling problem by defining a flow of interactions between components in the following sequence:

1. A user-written class, called a **Spider**, sends a request for a new page to an event-based **Engine** which acts as a hub connecting events to subscribers. Before the event is triggered, however, a list of filters called the **Spider Middleware** can be applied to post-process the request and handle any exceptions thrown by the user.
2. A **Scheduler** component listens to new requests in the hub and adds them to a priority queue to be fetched later.

3. When it's the turn of a request to be fetched, the Scheduler triggers an event in the Engine hub so a **Downloader** component can retrieve the resource.
4. Before the Downloader handles a request, however, a list of filters called the **Downloader Middleware** which may modify the request (e.g., modify the URL, add headers, etc.), respond to it without calling the Downloader or ignore it altogether.
5. Once the Downloader receives the request, it will download the resource and when it's ready it will send the response to the hub after applying the filters from the Downloader Middleware which may modify the response data.
6. The response is made available to the user-written Spider via its list of Spider Middleware filters which can redirect the data to specific handlers in the Spider. Those handlers will parse the document, retrieve relevant data and push it back to the Engine for further processing.
7. Before the extracted data is sent to the last step in the sequence, it will pass through a final layer of Spider Middleware filters which may add or change fields in the original data.
8. The **Item Pipelines** are a set of components that will listen to any data made available in the main hub and will function as the sink for the whole flow. They will be executed in a user-defined order and may be responsible for persisting the data either in memory or a hard drive storage unit.

Writing a new project in Scrapy

Scrapy comes with a CLI tool that helps users scaffold new projects, run them, add new Spiders, test them and interact with web resources in a Python interpreter environment to experiment with the code that will parse and extract data. In the following tutorial, I'll explore these steps.

```
$ scrapy startproject tutorial
$ tree .
tutorial/
|- scrapy.cfg
|- tutorial/
   |- __init__.py
   |- items.py
   |- middlewares.py
   |- pipelines.py
   |- settings.py
   |- spiders/
      |- __init__.py
```

The previous command will create a tree structure for a Scrapy project, the most relevant files to pay attention to are:

- **items.py** Includes classes representing Plain Old Python Objects that Spiders will send to the Item Pipelines.
- **middlewares.py** Includes the classes representing the Downloader and Spider Middleware filters.
- **pipelines.py** Includes the classes representing Item Pipelines.
- **settings.py** Settings for the Scrapy Architecture.

However, for this short tutorial, I won't be modifying any of these files and will just address the process of creating new spiders. I'll come back to them later in the next chapter when I talk about writing real-world Crawlers.

```
$ cd tutorial
$ scrapy genspider quoter quotes.toscrape.com
```

The previous commands will create a **basic** Spider named *quoter* that will crawl the https://quotes.toscrape.com website. A basic Spider functions as the one described in the previous section but there are other templates that one may use depending on the scenario:

- **XML Feed Spider** Given an XML resource, explores all the entities of a specific tag within the document.
- **CSV Feed Spider** Given a CSV resource, for each row in the document it will only explore a user given set of columns.
- **Crawl Spider** Defines a list of Regular Expression Rules that decide whether links in the current HTML page should be automatically followed or not for further crawling.
- **Sitemap Spider** A Crawl Spider that is able to interpret Sitemaps (https://sitemaps.org) and *robots.txt* files (https://www.robotstxt.org) to discover Web pages at the current path or domain while also defining handlers that will parse and extract data (scrape) from those resources.

To complete the basic setup, change the contents of *quoter.py* to match the following:

```
import scrapy

class QuoterSpider(scrapy.Spider):
    name = 'quoter'
    allowed_domains = ['quotes.toscrape.com']
    start_urls = ['https://quotes.toscrape.com']

    def parse(self, response):
        self.log('TITLE: "' + response.css('title::text').get() + '"')
```

This will simply log the `<title>` entity from the https://quotes.toscrape.com web page. To run the project, go to the source directory and execute the following command:

```
$ scrapy crawl quoter
$ ...
$ 2020-11-15 16:53:06 [quoter] DEBUG: TITLE: "Quotes to Scrape"
$ ...
```

The command will print many lines but one of them will be the title of the page we just extracted in our Spider above. In the next sections of this tutorial I'll keep enriching the basic Spider.

Extracting data via selectors

Scrapy extracts data from websites using XPath and CSS selectors. XPath selectors are useful when dealing with well-formed XML documents but for HTML pages, CSS selectors might be preferred as they can deal with bad formatting. To write selectors, it's convenient to use **SelectorGadget** (https://selectorgadget.com/) which can be installed as an add-on in Chrome. This tool allows the user to manually select elements on a page so that the tool can suggest a CSS rule to be used as a selector. Back in our tutorial project, when you visit the quotes to scrape website and enable the SelectorGadget plugin, you can now select a quote for extraction and see all the elements that will be covered by the suggested CSS selector:

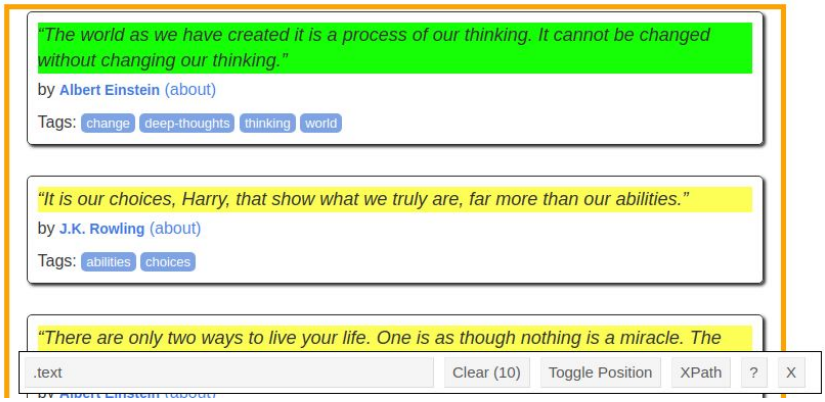

Fig 2. SelectorGadget suggests a CSS selector.

In the picture above, I selected the first quote (highlighted in green) and from that the tool suggested `.text` as the CSS rule highlighting in yellow other elements that would be covered by the same rule. You can further refine a CSS selector by clicking other non-highlighted elements to include similarly grouped items in the filter or by clicking a highlighted element to remove the groups they belong to from the filter. For example, in the tutorial page if you select one of the Tags below the quotes, the suggested rule will also include the tags listed in the Top Ten Tags list displayed on the right which you might not care about for this specific extraction. If we click any of the yellow highlighted tags from the right, the new suggested rule `.tags .tag` will no longer include the undesired tags in the filter. Assuming you want to extract a nested element inside a highlighted section you can shift click the item and the tool will suggest a more refined CSS rule. For example, `.text , .quote` will return the quote text inside each of the quote panels.

Playing with selectors

Before writing code to match the CSS selectors that we just generated, it might be useful to try them first in a playground environment. Scrapy offers a Python shell for that specific purpose, below I list commands that I used to figure out the CSS rules required to extract a list of quotes with their author and the tags associated with them.

```
$ scrapy shell 'http://quotes.toscrape.com'
>>> response.css('.quote')[0].css('.text::text').get()
'"The world as we have created it is a process of our thinking. It cannot be changed without changing
our thinking."'
>>> response.css('.quote')[0].css('.author::text').get()
'Albert Einstein'
>>> response.css('.quote')[0].css('.tag::text').getall()
['change', 'deep-thoughts', 'thinking', 'world']
```

The example above shows how to interact with a live page on a local environment to experiment with tentative selectors. The object `response` contains the HTML page that we can query to extract data from it, when your rule returns a selector or list of selectors (instead of raw text or HTML attributes), the data structure is recursive so you can keep applying further selectors to it to extract more refined data. In this example, I extract first a list of quotes via `.quote` and refine my search for the quote text, author and list of tags respectively. Notice that because the list of retrieved tags is exclusive to the current quote panel, the list of Top Ten Tags won't show up in this filter.

Note. CSS rules return a list of matching elements by default but sometimes you might expect this list to always contain exactly one element. In such cases you might want to call `get()` at the end of your extraction, otherwise `getall()` should return you a list of items.

Scraping quotes

Having figured out the CSS selectors you can go back to your Quoter Spider file and return every quote scraped from the website:

```
import scrapy

class QuoterSpider(scrapy.Spider):
    name = 'quoter'
    allowed_domains = ['quotes.toscrape.com']
    start_urls = ['https://quotes.toscrape.com']

    def parse(self, response):
        for quote in response.css('.quote'):
            yield {
                'quote': quote.css('.text::text').get(),
                'author': quote.css('.author::text').get(),
                'tags': quote.css('.tag::text').getall()
            }
```

Because you are now returning data from the `parse()` method you can now also run a command to automatically put the scraped data into a JSON file.

```
$ scrapy crawl quoter -o quotes.json
$ cat quotes.json
[
{"quote": "\u201cThe world as we have created it is a process of our thinking. It cannot be changed
without changing our thinking.\u201d", "tags": ["change", "deep-thoughts", "thinking", "world"],
"author": "Albert Einstein"},
{"quote": "\u201cIt is our choices, Harry, that show what we truly are, far more than our
abilities.\u201d", "tags": ["abilities", "choices"], "author": "J.K. Rowling"},
...
```

Hyperfields: Exploiting hypermedia

Let's assume that this application would benefit from scraping more information about the author. Luckily, this website readily provides such data through links on each quote that point to the author's biography page. Thanks to Scrapy's asynchronous framework, you may chain parsing handlers to construct an object whose fields exist on different pages at once. The following code will retrieve the author's birth date together with their name.

```
import scrapy

class QuoterSpider(scrapy.Spider):
    name = 'quoter'
    allowed_domains = ['quotes.toscrape.com']
    start_urls = ['https://quotes.toscrape.com']
```

```
    def parse(self, response):
        for quote in response.css('.quote'):
            item = {
                'quote': quote.css('.text::text').get(),
                'tags': quote.css('.tag::text').getall()
            }
            author_url = quote.css('.author + a::attr(href)').get()
            yield response.follow(
                    author_url,
                    cb_kwargs=item,
                    callback=self.parse_author
                  )

    def parse_author(self, response, quote, tags):
        yield {
            'author': {
                'name': response.css('.author-title::text').get(),
                'birthdate': response.css('.author-born-date::text').get()
            },
            'quote': quote,
            'tags': tags
        }
```

In this example, the result of calling the `follow()` method of response enqueues a new request to gather data about the author before being able to yield a complete data item to the Pipelines. Every call to `follow()` needs to include the callback method that will handle the response of the Web page once available. Additionally you may also pass along extra data to the callback as function arguments which I do here to relay the data scraped from the original quote. Feel free to run this code to validate that the new fields are printed.

Note. Instead of calling `follow()`, you can also yield a `Request` object directly but for that you would need to resolve first an absolute URI from the author's relative URL. Calling `follow()` saves you the trouble.

Note. Some applications or NoSQL databases benefit from denormalized data such as the one in this example. For normalized data, it might be more performant to first fetch the list of all possible authors and then map them to their quotes via a foreign key without the need to rely on hyperfields.

Crawling the entire website

So far you've been able to crawl the quotes from the first page of the website but in order to scrape further quotes from other pages you will need to follow the pagination links displayed at the bottom of the page. It's quite possible to write code that retrieves the pagination links and decides to enqueue further requests for scraping by calling `follow()` with a recursive callback that points to the original `parse()` method that handled the first page, however, for this tutorial I'll explore instead using the more specialized Crawl Spider that was designed for such tasks.

```
import scrapy
from scrapy.spiders import CrawlSpider, Rule
from scrapy.linkextractors import LinkExtractor

class QuoterSpider(CrawlSpider):
```

```
    name = 'quoter'
    allowed_domains = ['quotes.toscrape.com']
    start_urls = ['https://quotes.toscrape.com/page/1']

    rules = (
        Rule(LinkExtractor(allow=('page/\d+', ), deny=('tag', )), callback='parse', follow=True),
    )

    def parse(self, response):
        for quote in response.css('.quote'):
            item = {
                'quote': quote.css('.text::text').get(),
                'tags': quote.css('.tag::text').getall()
            }
            author_url = quote.css('.author + a::attr(href)').get()
            yield response.follow(
                    author_url,
                    cb_kwargs=item,
                    callback=self.parse_author,
                    dont_filter=True
                )

    def parse_author(self, response, quote, tags):
        yield {
            'author': {
                'name': response.css('.author-title::text').get(),
                'birthdate': response.css('.author-born-date::text').get()
            },
            'quote': quote,
            'tags': tags
        }
```

Highlighted in green are the changes made to the original Spider. Specifically, now I inherit from the Crawl Spider class and define rules that control which links in the current page will be crawled. It's important to note that I was able to use a Crawl Spider thanks to the fact that the pages I'm exploring have a URL with a consistent pattern, namely `'page/\d+'`, however, it might be the case that complex links that could not be generalized by Regular Expressions will need more elaborated rules that the Crawl Spider doesn't provide out-of-the-box. Crawl Spiders don't follow links unless they are explicitly allowed in a rule but sometimes a Regular Expression might filter in links that are unwanted, to disable them you can explicitly deny them in the Rule definition. In this example, it just so happens that tags link to pages with the URL `'tag/tag_name/page/\d+'` will be covered by the allowed expressions which is why I'm explicitly disabling them.

Note. Scrapy records by default which URLs have been previously visited to avoid duplicates or infinite loops in the case of Crawl Spiders. This is why I'm able to crawl every single page on the website even though the paginator will sometimes point to previously visited pages (the previous one). This works great for most cases but when dealing with denormalized data, such as when exploring hyperfields, it is by design that you will have to explore duplicate pages. To disable the default behavior, I use dont_filter and set it to True.

Testing your Spiders

As it's the case for almost every piece of code in Python, you can write a Unit Test to validate that your code matches your expectations. Scrapy code is no exception although you might also want to set up a test environment in which web requests are intercepted and fake data is returned so that static expectations can match consistently

in your tests. Scrapy, however, offers one more option called **Spider Contracts** that you might find helpful for some basic scenarios considering this feature is not particularly powerful (or consistent unless you set up a fake server like indicated before). Contracts are annotations you add to your Spider code and that you verify by running:

```
$ scrapy check
----------------------------------------------------------------------
Ran 3 contracts in 1.098s

OK
```

```
    def parse(self, response):
        """ This function parses a page with quotes and follows with a list of requests to gather
        further information about the author of each quote.

        @url https://quotes.toscrape.com/page/1
        @returns requests 10 10
        """
        pass

    def parse_author(self, response, quote, tags):
        """ This function parses an author page and returns a quote from previously parsed data.

        @url https://quotes.toscrape.com/author/Albert-Einstein/
        @cb_kwargs {
            "quote": "If it's on the Internet, it must be true.",
            "tags": ["misleading", "internet"]
        }
        @returns items 1 1
        @scrapes author quote tags
        """
        pass
```

In the example above I have removed code that is unrelated to Contracts but on your end you don't need, instead just add the triple quoted comments that represent the validation checks of each method.

Spider Contracts use docstring attributes to set the test expectations. Scrapy provides out-of-the-box rules to validate basic scenarios but you can write your own if you find it easier than writing a custom test. You can refer to the documentation[1] for further details. Regarding the contracts provided by Scrapy, these are the most useful:

- **url** Mandatory field for the web page that will be parsed and processed by the test engine.
- **returns <requests|items>** Sets lower and upper bounds (upper bound is optional) for the number of requests or items that the function should return.
- **scrapes** For functions that return items, lists the fields that should always exist in every output record. Doesn't support nested fields.
- **cb_kwargs** For functions that receive more arguments than just the page response (e.g., in the case of hyperfields).

[1] https://docs.scrapy.org/en/latest/topics/contracts.html#custom-contracts

Writing real-world Crawlers in Scrapy

As mentioned in the introduction, writing Web Crawlers for complex scenarios poses several challenges. In this chapter I'll describe how you may address them using Scrapy.

Parsing media in formats other than XHTML

Scrapy provides Crawl Spiders that handle HTML as well as Feed Spiders that handle XML and CSV data. If you have specific formats that you need to support (e.g., PDF), Scrapy or its community don't provide any specific tool so you will have to write your own parser, e.g., using a PDF parsing library.

Complying with Robots.txt

Servers might define a list of sites which they prefer not be crawled in a robots.txt file accessible at the root path of a website, Scrapy consults this list and abides by it but if you find a compelling reason to avoid this behavior, you can prevent Scrapy from complying with robots.txt by setting this variable in the *settings.py* file created at the beginning of the previous chapter:

```
ROBOTSTXT_OBEY = False
```

Interacting with slow or overloaded servers

Scrapy implements an auto throttling algorithm to "play nicely" with servers that may be experiencing an overload of requests. This algorithm works by setting an initial delay to all requests and then dynamically change it by assessing how fast or slow a server is responding with 200 (OK) responses, the faster the server the smaller the delay and vice-versa. Settings for this algorithm can be configured in the *settings.py* file:

- **AUTOTHROTTLE_ENABLED** Set to True if auto throttling is to be enabled.
- **AUTOTHROTTLE_START_DELAY** The starting delay for all requests before any dynamic adjustment.
- **AUTOTHROTTLE_MAX_DELAY** The maximum delay for every request that any dynamic adjustment should not exceed.
- **AUTOTHROTTLE_TARGET_CONCURRENCY** The maximum number of concurrent requests to send a server.
- **AUTOTHROTTLE_DEBUG** Logs debug information if set to True.
- **CONCURRENT_REQUESTS_PER_DOMAIN** Similar to target concurrency but this is a global setting not specific to auto throttling.
- **CONCURRENT_REQUESTS_PER_IP** Same as above but per client IP and not per domain. Useful to avoid bot detection mechanisms.
- **DOWNLOAD_DELAY** The minimum delay for every request that any dynamic adjustment should not lower.

Bypassing bot detection mechanisms

There are extensions to Scrapy that allow it to rotate client IPs and user-agents so that servers are unable to detect them as bots. From the server's perspective, each crawling request might come from different user agents or IPs, though it's possible that more sophisticated servers might still be able to detect this mechanism.

scrapy-user-agents is a plugin that can be installed via *pip install* that when configured in *settings.py* file will rotate user agents.

```
DOWNLOADER_MIDDLEWARES = {
    'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware': None,
    'scrapy_user_agents.middlewares.RandomUserAgentMiddleware': 400,
}
```

The above configuration disables the default Scrapy mechanism that sets a user agent and replaces it for the one from the plugin which rotates user-agents. The number next to the setting key is the priority order in which the filter will be applied, lower numbers will be applied first.

scrapy-proxy-pool is another plugin to be installed in a similar way than the previous one and requires the following configuration:

```
PROXY_POOL_ENABLED = True
DOWNLOADER_MIDDLEWARES = {
    # ...
    'scrapy_proxy_pool.middlewares.ProxyPoolMiddleware': 610,
    'scrapy_proxy_pool.middlewares.BanDetectionMiddleware': 620,
    # ...
}
```

Lastly, there is a middleware that is supposedly able to bypass Captcha tests but I was not able to test it or evaluate it for this report. I still include it for reference, it's called captcha-middleware[2] and relies on a Text Recognition Library (Tesseract) to recognize text-based Captchas. It's important to keep in mind that Captchas nowadays rely on object matching (e.g., click on every bicycle image) than on character recognition but this library might still be helpful for some situations.

Deploying to the Cloud

Web Crawling and Scraping is resource intensive so it makes sense to run your Spiders in a Cloud environment. There is such a commercial option for Scrapy called Scrappinghub (https://www.scrapinghub.com). You can simply run a set of simple commands to upload to their servers the same kind of projects that I showed in the tutorial. Additionally, features from your local Scrapy such as remembering previously visited sites are also made available on Scrappinghub via shared storage services called **Frontiers**. Once your project is ready, you can start jobs on a schedule (or at whim) to crawl and scrape for Web data. The specifics on how to use the tool are beyond the scope of this report, however.

Keeping fresh data from sites that update frequently

For websites that update on a fixed schedule (potentially defined in the Sitemaps file), you can match such schedule in the Web UI of Scrappinghub, however, if you need to adjust this schedule dynamically (based on site popularity or other factors) you will need to write your own scheduler. Scrapphinghub exposes a robust REST API that makes writing such a tool possible but describing this API goes beyond the scope of this report.

[2] https://github.com/owen9825/captcha-middleware

Conclusions

Scrapy is a robust framework that supports a great number of Crawling requirements as shown in this paper and because of its popularity and large community there are many options and tools for users that want to write Crawlers for complex sites that require lots of resources and specific configurations. While it might not cover every possible scenario, it definitely provides you with the tools to write your own plugin so that you can enrich the already robust pipeline. For startups or users that need to scrape big amounts of data from the Web this tool is definitely ideal and recommended from my point of view as it's simple to learn, use and extend. The same recommendation goes for large corporations in case they don't have themselves the infrastructure to run Web Crawling tasks.