Published in Velotio Perspectives

Velotio Technologies Follow
Feb 11, 2019 · 11 min read · Listen

Web Scraping: Introduction, Best Practices & Caveats

Introduction:

Web scraping is a process to crawl various websites and extract the required data using spiders. This data is processed in a data pipeline and stored in a structured format. Today, web scraping is widely used and has many use cases:

- Using web scraping, Marketing & Sales companies can fetch lead-related information.
- Web scraping is useful for Real Estate businesses to get the data of new projects, resale properties, etc.

The process of web scraping usually involves spiders, which fetch the HTML documents from relevant websites, extract the needed content based on the business logic, and finally store it in a specific format. This blog is a primer to build highly scalable scrappers. We will cover the following items:

1. **Ways to scrape**: We'll see basic ways to scrape data using techniques and frameworks in Python with some code snippets.
2. **Scraping at scale**: Scraping a single page is straightforward, but there are challenges in scraping millions of websites, including managing the spider code, collecting data, and maintaining a data warehouse. We'll explore such challenges and their solutions to make scraping easy and accurate.
3. **Scraping Guidelines**: Scraping data from websites without the owner's permission can be deemed as malicious. Certain guidelines need to be followed to ensure our scrappers are not blacklisted. We'll look at some of the best practices one should follow for crawling.

So let's start scraping.

Different Techniques for Scraping

Here, we will discuss how to scrape a page and the different libraries available in Python.

Note: Python is the most popular language for scraping.

1. **1. Requests — HTTP Library in Python**: To scrape the website or a page, first find out the content of the HTML page in an HTTP response object. The requests library from Python is pretty handy and easy to use. It uses urllib inside. I like 'requests' as it's easy and the code becomes readable too.

```
#Example showing how to use the requests library
import requests
r = requests.get("https://velotio.com") #Fetch HTML Page
```

requests.sh hosted with ❤ by GitHub view raw use the *requests* library to fetch an HTML page and then use the *BeautifulSoup* to parse that page. In this example, we can easily fetch the page title and all links on the page. Check out the documentation for all the possible ways in which we can use BeautifulSoup.

```python
from bs4 import BeautifulSoup
import requests
r = requests.get("https://velotio.com") #Fetch HTML Page
soup = BeautifulSoup(r.text, "html.parser") #Parse HTML Page
print "Webpage Title:" + soup.title.string
print "Fetch All Links:" soup.find_all('a')
```

documentation.js hosted with ❤ by GitHub view raw

3. **Python Scrapy Framework**:

Scrapy is a Python-based web scraping framework that allows you to create different kinds of spiders to fetch the source code of the target website. Scrapy starts crawling the web pages present on a certain website, and then you can write the extraction logic to get the required data. Scrapy is built on the top of Twisted, a Python-based asynchronous library that performs the requests in an async fashion to boost up the spider performance. Scrapy is faster than BeautifulSoup. Moreover, it is a framework to write scrapers as opposed to BeautifulSoup, which is just a library to parse HTML pages.

Here is a simple example of how to use Scrapy. Install Scrapy via pip. Scrapy gives a shell after parsing a website:

```
$ pip install scrapy #Install Scrapy"
$ scrapy shell https://velotio.com
In [1]: response.xpath("//a").extract() #Fetch all a hrefs
```

install_scrapy.sh hosted with ❤ by GitHub view raw

Now let's write a custom spider to parse a website.

```
$cat > myspider.py <import scrapy
```

```
def parse(self, response):
for title in response.css('h2.entry-title'):
yield {'title': title.css('a ::text').extract_first()}
EOF
scrapy runspider myspider.py
```

custom_spider.sh hosted with ❤ by GitHub view raw

That's it. Your first custom spider is created. Now. let's understand the code.

- **name**: Name of the spider. In this case, it's "blogspider".
- **start_urls**: A list of URLs where the spider will begin to crawl from.
- **parse**(self, response): This function is called whenever the crawler successfully crawls a URL. The response object used earlier in the Scrapy shell is the same response object that is passed to the parse(..).

When you run this, Scrapy will look for start URL and will give you all the divs of the **h2.entry-title** class and extract the associated text from it. Alternatively, you can write your extraction logic in a parse method or create a separate class for extraction and call its object from the parse method.

You've seen how to extract simple items from a website using Scrapy, but this is just the surface. Scrapy provides a lot of powerful features for making scraping easy and efficient. Here is a tutorial for Scrapy and the additional documentation for LinkExtractor by which you can instruct Scrapy to extract links from a web page.

4. Python lxml.html library: This is another library from Python just like BeautifulSoup. Scrapy internally uses lxml. It comes with a list of APIs you can use for data extraction. Why will you use this when Scrapy itself can extract the data? Let's say you want to iterate over the 'div' tag and perform some operation on each tag present under "div", then you can use this library which will give you a list of 'div' tags. Now you can simply iterate over them using the iter() function and traverse each child tag inside the parent div tag. Such traversing operations are difficult in scraping. Here is the documentation for this library.

Challenges while Scraping at Scale

1. **Data warehousing**: Data extraction at a large scale generates vast volumes of information. Fault-tolerant, scalability, security, and high availability are the must-have features for a data warehouse. If your data warehouse is not stable or accessible then operations, like search and filter over data would be an overhead. To achieve this, instead of maintaining own database or infrastructure, you can use Amazon Web Services (AWS). You can use RDS (Relational Database Service) for a structured database and DynamoDB for the non-relational database. AWS takes care of the backup of data. It automatically takes a snapshot of the database. It gives you database error logs as well. This blog explains how to set up infrastructure in the cloud for scraping.

2. **Pattern Changes:** Scraping heavily relies on user interface and its structure, i.e., CSS and Xpath. Now, if the target website gets some adjustments then our scraper may crash completely or it can give random data that we don't want. This is a common scenario and that's why it's more difficult to maintain scrapers than writing it. To handle this case, we can write the test cases for the extraction logic and run them daily, either manually or from CI tools, like Jenkins to track if the target website has changed or not.

3. **Anti-scraping Technologies:** Web scraping is a common thing these days, and every website host would want to prevent their data from being scraped. Anti-scraping technologies would help them in this. For example, if you are hitting a particular website from the same IP address on a regular interval then the target website can block your IP. Adding a captcha on a website also helps. There are methods by which we can bypass these anti-scraping methods. For e.g., we can use proxy servers to hide our original IP. There are several proxy services that keep on rotating the IP before each request. Also, it is easy to add support for proxy servers in the code, and in Python, the Scrapy framework does support it.

4. **JavaScript-based dynamic content:** Websites that heavily rely on JavaScript and Ajax to render dynamic content, makes data extraction difficult. Now, Scrapy and related frameworks/libraries will only work or extract what it finds in the HTML document. Ajax calls or JavaScript are executed at runtime so it can't scrape that. This can be handled by rendering the web page in a headless browser such as Headless Chrome, which essentially allows running Chrome in a server

5. **Honeypot traps:** Some websites have honeypot traps on the webpages for the detection of web crawlers. They are hard to detect as most of the links are blended with background color or the display property of CSS is set to none. To achieve this requires large coding efforts on both the server and the crawler side, hence this method is not frequently used.

6. **Quality of data:** Currently, AI and ML projects are in high demand and these projects need data at large scale. Data integrity is also important as one fault can cause serious problems in AI/ML algorithms. So, in scraping, it is very important to not just scrape the data, but verify its integrity as well. Now doing this in real-time is not possible always, so I would prefer to write test cases of the extraction logic to make sure whatever your spiders are extracting is correct and they are not scraping any bad data

7. **More Data, More Time:** This one is obvious. The larger a website is, the more data it contains, the longer it takes to scrape that site. This may be fine if your purpose for scanning the site isn't time-sensitive, but that isn't often the case. Stock prices don't stay the same over hours. Sales listings, currency exchange rates, media trends, and market prices are just a few examples of time-sensitive data. What to do in this case then? Well, one solution could be to design your spiders carefully. If you're using Scrapy like framework then apply proper LinkExtractor rules so that spider will not waste time on scraping unrelated URLs.

You may use multithreading scraping packages available in Python, such as Frontera and Scrapy Redis. Frontera lets you send out only one request per domain at a time but can hit multiple domains at once, making it great for parallel scraping. Scrapy Redis lets you send out multiple requests to one domain. The right combination of these can result in a very powerful web spider that can handle both the bulk and variation for large websites.

8. Captchas: Captchas is a good way of keeping crawlers away from a website and it is used by many website hosts. So, in order to scrape the data from such websites, we need a mechanism to solve the captchas. There are packages, software that can solve the captcha and can act as a middleware between the target website and your spider. Also, you may use libraries like Pillow and Tesseract in Python to solve the simple and that may introduce scraping of millions of websites. Now, you can imagine the size of the code and the deployment. We can't run spiders at this scale from a single machine. What I prefer here is to dockerize the scrapers and take advantage of the latest technologies, like AWS ECS, Kubernetes to run our scraper containers. This helps us keeping our scrapers in high availability state and it's easy to maintain. Also, we can schedule the scrapers to run at regular intervals

Scraping Guidelines/ Best Practices:

1. **Respect the robots.txt file:** Robots.txt is a text file that webmasters create to instruct search engine robots on how to crawl and index pages on the website. This file generally contains instructions for crawlers. Now, before even planning the extraction logic, you should first check this file. Usually, you can find this at the website admin section. This file has all the rules set on how crawlers should interact with the website. For e.g., if a website has a link to download critical information then they probably don't want to expose that to crawlers. Another important factor is the frequency interval for crawling, which means that crawlers can only hit the website at specified intervals. If someone has asked not to crawl their website then we better not do it. Because if they catch your crawlers, it can lead to some serious legal issues.

2. **Do not hit the servers too frequently:** As I mentioned above, some websites will have the frequency interval specified for crawlers. We better use it wisely because not every website is tested against the high load. If you are hitting at a constant interval then it creates huge traffic on the server-side, and it may crash or fail to serve other requests. This creates a high impact on user experience as they are more important than the bots. So, we should make the requests according to the specified interval in robots.txt or use a standard delay of 10 seconds. This also helps you not to get blocked by the target website.

3. **User Agent Rotation and Spoofing:** Every request consists of a User-Agent string in the header. This string helps to identify the browser you are using, its version, and the platform. If we use the same User-Agent in every request then it's easy for the target website to check that request is coming from a crawler. So, to make sure we do not face this, try to rotate the User and the Agent between the requests. You

4. **Disguise your requests by rotating IPs and Proxy Services:** We've discussed this in the challenges above. It's always better to use rotating IPs and proxy service so that your spider won't get blocked.

5. **Do not follow the same crawling pattern:** Now, as you know many websites use anti-scraping technologies, so it's easy for them to detect your spider if it's crawling in the same pattern. Normally, we, as a human, would not follow a pattern on a particular website. So, to have your spiders run smoothly, we can introduce actions like mouse movements, clicking a random link, etc, which gives the impression of your spider as a human.

6. **Scrape during off-peak hours:** Off-peak hours are suitable for bots/crawlers as the traffic on the website is considerably less. These hours can be identified by the geolocation from where the site's traffic originates. This also helps to improve the crawling rate and avoid the extra load from spider requests. Thus, it is advisable to schedule the crawlers to run in the off-peak hours.

7. **Use the scraped data responsibly:** We should always take the responsibility of the scraped data. It is not acceptable if someone is scraping the data and then republish it somewhere else. This can be considered as breaking the copyright laws and may lead to legal issues. So, it is advisable to check the target website's **Terms of Service** page before scraping.

8. **Use Canonical URLs:** When we scrape, we tend to scrape duplicate URLs, and hence the duplicate data, which is the last thing we want to do. It may happen in a single website where we get multiple URLs having the same data. In this situation, duplicate URLs will have a canonical URL, which points to the parent or the original URL. By this, we make sure, we don't scrape duplicate contents. In frameworks like Scrapy, duplicate URLs are handled by default.

9. **Be transparent:** Don't misrepresent your purpose or use deceptive methods to gain access. If you have a login and a password that identifies you to gain access to a source, use it. Don't hide who you are. If possible, share your credentials.

Conclusion:

- Maintenance of data and spiders at scale is difficult. Use Docker/ Kubernetes and public cloud providers, like AWS to easily scale your web-scraping backend.
- Always respect the rules of the websites you plan to crawl. If APIs are available, always use them first.

****

This post was originally published on ***Velotio Blog*****.**

Velotio Technologies is an outsourced software product development partner for technology startups and enterprises. We specialize in enterprise B2B and SaaS product development with a focus on artificial intelligence and machine learning, DevOps, and test engineering. We combine innovative ideas with business expertise and cutting-edge technology to drive business success for our customers.

Interested in learning more about us? We would love to connect with you on our ***Website****,* ***LinkedIn*** *or* ***Twitter****.*

****