Codementor Team FOLLOW
On-Demand Marketplace for Software Developers

How to Scrape an AJAX Website using Python

Published Dec 19, 2016

Web scraping is a technique used to retrieve information from a web page using software. There are many tools to do web scraping with Python, some of them are:

- Scrapy
- Sky
- Beautiful Soup
- Requests

The problem with most of these tools is that they only retrieve the static HTML that comes from the server and not the dynamic part which is rendered using JavaScript.

They are some options to overcome this problem:

1. **Automated browsers**

Automated web browsers like Selenium or Splash are full browsers that run "headless." Setting this up is a bit complex. There are solutions out there to mitigate this, such as docker.
Setting these up is beyond the scope of this document, and so we will focus our solution on the second option below.

2. **Intercept AJAX calls**

   Try to intercept the AJAX calls from the page and reproduce/replay them.

Scope of this tutorial

This tutorial will teach you how to catch AJAX calls and reproduce them using the `requests` library and the Google Chrome browser. While frameworks like `scrapy` provide a more robust solution for web scraping, it is not necessary for all cases. AJAX calls are mostly done against an API that returns a JSON object which can be easily handled by the `requests` library. This tutorial can be done with any other browser like Firefox — the process is the same, the only thing that changes is the dev tools user interface.

For this tutorial, we will use a real example: retrieving all the Wholefoods store locations from their website.

Let's start

The first thing you need to know is that trying to reproduce an AJAX call is like using an undocumented API, so you must look at the call the pages make. Go to the site and play with it, navigate through some pages and see how some info is rendered without the need of going to another URL. After you finish playing, come back here to start the scraping.

Before getting into coding, we first need to know how the page works. For

that, we are going to navigate through to the pages containing the store information.

First, let's get to the find store section of the site:

Let's limit our scraping to the stores located in the US by state:

We now see stores by US state. Select any state and the page will render the store's info located there. Try it out and then come back. There is also the option to see stores by page, this is the option that we'll be using to scrap the page.

As you can see, every time you select a state or page the website renders new stores, replacing the old ones. This is done using an AJAX call to a server asking for the new stores. Our job now is to catch that call and reproduce it. For that, open the Chrome DevTools console, and go to the network section and XHR subsection:

XHR (XMLHttpRequest) is an interface to do HTTP and HTTPS requests, so it's most likely that the ajax request would be shown here.

Now, while monitoring the network, select the second page to see what happens. You should see something like this:

If you double click the AJAX call, you will see that there are lots of info there about the stores. You can preview the requests, too. Let's do that to see if the info that we want is there. The AJAX call doesn't always have the name "AJAX", it could have any name, so you must look what data comes in every call of the XHR subsection.

A response arrives in a JSON format. It has 3 elements, and the info that we want is in the last one. This element contains a data key that has the HTML that is inserted in the page when a page is selected. As you will note, the HTML is in a really long line of text. You can use something like Code Beautify to "prettify" it. The block of code that corresponds to every store is in this gist. In this block is everything that we need.

Now we look at the headers section to see all the info that goes to the server in the call.

Here we see the request URL and that the request method is post. Lets look now at the form data that is send to the server.

As you can see, a lot of data is sent to the server, but don't worry — not all of it is needed. To see what data is really needed, you can open a console and do various post requests to the request URL using the requests library, I already did that for us and highlighted the data that is needed.

Writing your own website scraper

Now that we know how the page works and deciphered the AJAX call, it is time to start writing our scraper. All the code that's written here is available in this github repository.

We will be using the lxml CSS selector to extract the info that we need. We will extract the following:

- Store name
- Full address
- Phone number
- Opening hours

Let's first create a project and cd to it.

```
$ mrkdir wholefoods-scraper
$ cd wholefoods-scraper
```

We should create a virtualenv.

```
$ virtualenv venv
$ source venv/bin/activate
```

Now we can install the requests library and make a Python file for the scraper.

```
$ pip install requests
$ pip install lxml
$ pip install cssselect
$ touch scraper.py
```

Now open the Python file with your favorite editor. Let's begin creating our scraper with a class and making a function to do replicate the AJAX call:

```
# Importing the dependencies
# This is needed to create a lxml object that uses the css selector
 from lxml.etree import fromstring

# The requests library
import requests

class WholeFoodsScraper:

  API_url = 'http://www.wholefoodsmarket.com/views/ajax'
    scraped_stores = []

    def get_stores_info(self, page):

        # This is the only data required by the api
        # To send back the stores info
        data = {
        'view_name': 'store_locations_by_state',
        'view_display_id': 'state',
        'page': page
        }
        # Making the post request
```

```
        # Making the post request
        response = requests.post(self.API_url, data=data)

        # The data that we are looking is in the second
        # Element of the response and has the key 'data',
        # so that is what's returned
        return response.json()[1]['data']
```

Now that we have a function that retrieves the data from the stores, let's make one to parse that data using the CSS selector, look at the gist if you get lost.

```
import ....

class WholeFoodsScraper:
  # Same ...
  # Array to store the scraped stores
  scraped_stores = []

# get_store_info function

def parse_stores(self, data):
  # Creating an lxml Element instance
      element = fromstring(data)

      for store in element.cssselect('.views-row'):
          store_info = {}

          # The lxml etree css selector always returns a list, so
          # just the first item
          store_name = store.cssselect('.views-field-title a')[0].
          street_address = store.cssselect('.street-block div')[0]
          address_locality = store.cssselect('.locality')[0].text
          address_state = store.cssselect('.state')[0].text
          address_postal_code = store.cssselect('.postal-code')[0]
          phone_number = store.cssselect('.views-field-field-phone

          try:
              opening_hours = store.cssselect('.views-field-field-
          except IndexError:
              # Stores that doesn't have opening hours are closed
              # This is found while debugging, so don't worry if y
```

```
            # This is found while debugging, so don't worry if y
            # run a scraper
            opening_hours = 'STORE CLOSED'
            continue

        full_address = "{}, {}, {} {}".format(street_address,
                                              address_locality,
                                              address_state,
                                              address_postal_cod

        # now we add all the info to a dict
        store_info = {
                    'name': store_name,
                    'full_address': full_address,
                    'phone': phone_number,
                    'hours': opening_hours
                    }

        # We add the store to the scraped stores list
        self.scraped_stores.append(store_info)
```

Now we just need a function to call the `get_stores_info` for every one of the 22 pages that has the Wholefoods stores by US state site and then parse the data with `parse_stores`. We will also make a small function to save all the scraped data to a JSON file.

```
import json
import ... All stays the same

class WholeFoodsScraper:
  # Same ...

  def run(self):
        for page in range(22):
            # Retrieving the data
            data = self.get_stores_info(page)
            # Parsing it
            self.parse_stores(data)
            print('scraped the page' + str(page))

        self.save_data()
```

```
        self.save_data()

    def save_data(self):
        with open('wholefoods_stores.json', 'w') as json_file:
            json.dump(self.scraped_stores, json_file, indent=4)
```

Now our scraper is ready! We just need to run it. Add this to the end of the python file:

```
if __name__ == '__main__':
  scraper = WholeFoodsScraper()
  scraper.run()
```

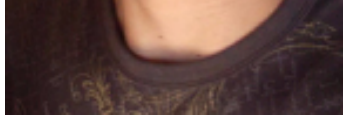
You can see the results in the `wholefoods_stores.json` file.

Conclusion

Now you must be asking, "*When should I use an automated browser and, well, should I try to reproduce the AJAX calls?*" The answer is simple: always try to reproduce the AJAX calls before trying something more complicated or heavy like using an automated browser — try this simpler and lighter approach!

Similarly, you can also try building your own web scraper using Node.js.

Author's Bio



Julio Alejandro is a freelance software developer and chemical engineer from Venezuela with growing skills. He loves doing web development, web scraping, and automation! He works mostly doing server-side programming with Python/Django/Flask. He also does front-end development with JavaScript frameworks like Vue or React.

Ajax | Web scraping | Scraper | Python

Enjoy this post? Give **Codementor Team** a like if it's helpful.

Codementor Team

On-Demand Marketplace for Software Developers

Our team is obsessed with learning about new technologies. We post about development learning, step-by-step guides, technical tutorials, as well as Codementor community announcements to help keep you up-to-date.

12 Replies

Leave a reply

Bruce Dailey Jr 2 months ago

Hello Julio,
I enjoyed your article. I have been working with the code you provided, but have been unsuccessful at getting it to work. The part I am stuck on is the response only returns the first element [0]. The

Show more

Reply

Bruce Dailey Jr 2 months ago

Actually just resolved part of the issue, the url was set to be http and now requires https. Currently getting a no name error on the third record, from the lxml parser.

Reply

Kevin Millan 6 months ago

would have been even better that you included screenshots of the things we are suppossed to be seeing while we go. For someone who doesn't know html, it can be quite difficult to follow along what

Show more

Reply

James De Leon a year ago

A detailed tutorial compared to the others I've read.

But sorry one question, I tried this code on an api_url with a query string parameter and it doesn't seem to work.

Reply