Scraping Guide

This guide is meant to help get you to a place where you feel comfortable enough to experiment with your own scraping. One of the best ways to truly learn how to do this stuff is to try and do it on your own. Every webpage is different, with different APIs, and different policies around scraping *(a lot of the time even if a company doesn't want you to, it doesn't mean you can't. It just means you have to work harder for it)*. There is no one true method to scraping but practicing on your own and feeling out different webpages can you help you understand what to look for and how to get the data you are interested in.

I would not worry about reading everything placed in this guide, but I hope that a lot of it can serve as a helpful quick reference if certain topics aren't fully understood. Scraping is an art not a science. While you can understand why certain things work or don't work the easiest way to make progress is to try.

Happy Hunting

> This guide pertains to scraping locally, not server side scraping. This should serve as a good foundation for getting started with scraping concepts. If you are looking to automate this work and run it on a server you can still use a lot of these concepts, you will just need to figure out ways to ensure they are run and perform some more error checking before you employ this in a production environment.

Table of Contents

There are not actual pages or anything but in order of items that will be covered

Common Vocab

Payload/Body: The content we are sending to the server. You could also interpret this as the main information we are receiving from a server.

Domain: Domains serve as a way to make machine IP addresses human readable. Something like `google.com` gets converted into an IP address like `192.10.10.1` (not the actual IP). By a DNS server.

Endpoint/URL: A endpoint can be thought of as a particular resource on a server. Something like `youtube.com/@somechannel`. We are requesting a specific resource hosted on the `youtube.com` domain. A URL is a "Universal Resource Locator" in this instance the endpoint/url would be `/@somechannel`.

> I know there is not a lot here so if there are words you are unclear on that I did not define please feel free to ask.

HTTP/S Basics

Some key understandings for scraping can be derived from the basics of HTTP/S protocol.

HTTP vs HTTPS

Hyper Text Transfer Protocol is the primary protocol used to transfer information over the internet. There are many kinds of protocols for data transferring. Some you may have seen or heard of before SSH (Secure Shell), and FTP (File Transfer Protocol). Protocols outline a standard that enables computers to communicate in an expected way that allows data to be transferred consistently.

The one we really care about here is HTTP. **HTTPS** is just a secured version of HTTP that encrypts the outgoing data before it reaches the open web so that information can't be extracted (like your passwords or other sensitive information).

For our purposes the distinction does not fully matter (we do not have to encrypt anything on our end even when using HTTPS). Most times as well if a request can be sent as HTTPS it can be sent as HTTP(typically). However you run the risk of an unexpected party intercepting these packets.

HTTP Requests Methods

There are 3 main HTTP requests (there are others but I think most people use these three consistently)

GET

GET requests are when we are asking the server for a resource. Typically we are not sending information with these requests, and there is no way to send a payload to the server when initiating a GET request. By default when you go to a webpage on your browser your browser is initiating a

GET request on that main resource i.e `https://google.com` sends a GET request and google sends back HTML to load their main page.

POST

POST requests are when we want to send data to the server / a particular resource on a server. POST requests allow us to send payloads to a server. Typically a POST request will result in some change of state. A lot of login requests are POST requests because we want to send our password information in the body (this ties into the HTTPS encryption)

DELETE

DELETE requests are somewhat similar to a GET request. The difference is primarily semantic or for organization. You could delete a specified resource with a GET but it is helpful for developers to distinguish between the two.

There are other request methods

There are 9 total request methods. Most of them are rare. The one you might see is a PUT but this shares a lot of similarities with a POST so I think most people just use POST.

HTTP Headers

Headers are used to convey important information about a request to the server. These help the server know how to create a response appropriately to the request. Think of headers as meta-data for a request.

There are a lot of headers that get used across requests, however not all are necessary to include in a request.

For a fully comprehensive list of headers and common options see this link:
https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers

Commonly required headers

Content-Type

The `Content-Type` header defines the content type we are sending to the server. Often times this needs to match what the server is expecting. We only care about this argument when we are making a POST/PUT request and need to send a body with our request.

Common argument in scraping will be `application/json` because most APIs will take in a dictionary/json in order to efficiently perform an action.

Authorization

The `Authorization` header is where we can place auth tokens to make requests to protected endpoints. You will typically have to go through some kind of authorization flow to acquire a token which can be different per website / api. Once you have this token you will need it in order to access protected endpoints.

HTTP Payloads

This is dependent on your specified content type. Typically for scraping we will mostly be using JSONs which python / javascript can handle easily and don't require a lot of set up on our end to accomplish.

HTTP Status Codes

These are fairly important to know here is a good reference https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status

For scraping we are mainly concerned with the 200 codes, and the 400 codes.

100 - 199 : Informational

These are less common to see for our purposes. They don't indicate a failure but rather that there is sort of more to be done. For example `100` indicates a "continue" which means the initiator should continue their request. Again these are more rare to see for our purposes

200 - 299 Success Codes

These are important for scraping. The main one we really want to see is a `200`. `200` indicates a successful request, and means that the data you requested is sitting in the response the server has sent you. Job is done essentially. Other `20X` responses might not indicate a bad thing, but typically for scraping your goal is to collect data and the other status codes might imply your data is not right where you wanted it.

300 - 399 Redirection Codes

These codes mean that your request has been redirected to the correct location. This is typically used if something has moved and they don't want to make the old URL invalid. Not crazy important for us but if you see these it can mean that there might be a more direct path to your data

400 - 499 Client Error Codes

These codes are important, however are not always set up in a truly helpful way to debug information. Everyone is familiar with the `404` code from seeing the placeholder page that shows

up when you type in a bad url.

Technically they have meanings, but this is entirely up to the website developers to incorporate meaningfully. Adding too much meaning can actually lead to people doing things you don't want them to do.

Typically though a `400` means your request was malformed.
A `401` means you probably needed some kind of authorization
`403` can mean your auth token does not grant you access to that resource, or that you are not authed at all (akin to a `401`)

`405` can actually be helpful because this typically means you sent the wrong method i.e a GET when you should have sent a POST or vice versa.

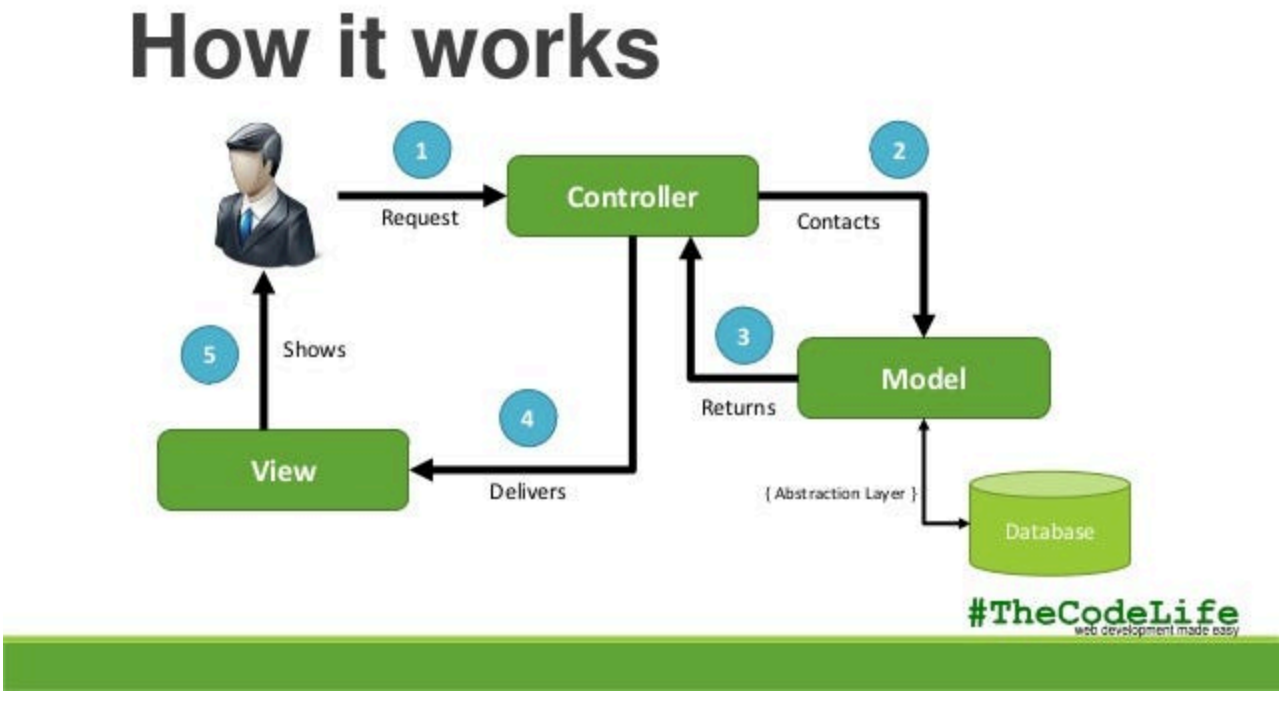
500 - 599 Server Error Codes

These mean something went wrong server side and the server was unable to process your request. If you see these most of the time they mean you broke something / the back end code broke.

Understanding Webpage and Application Flows

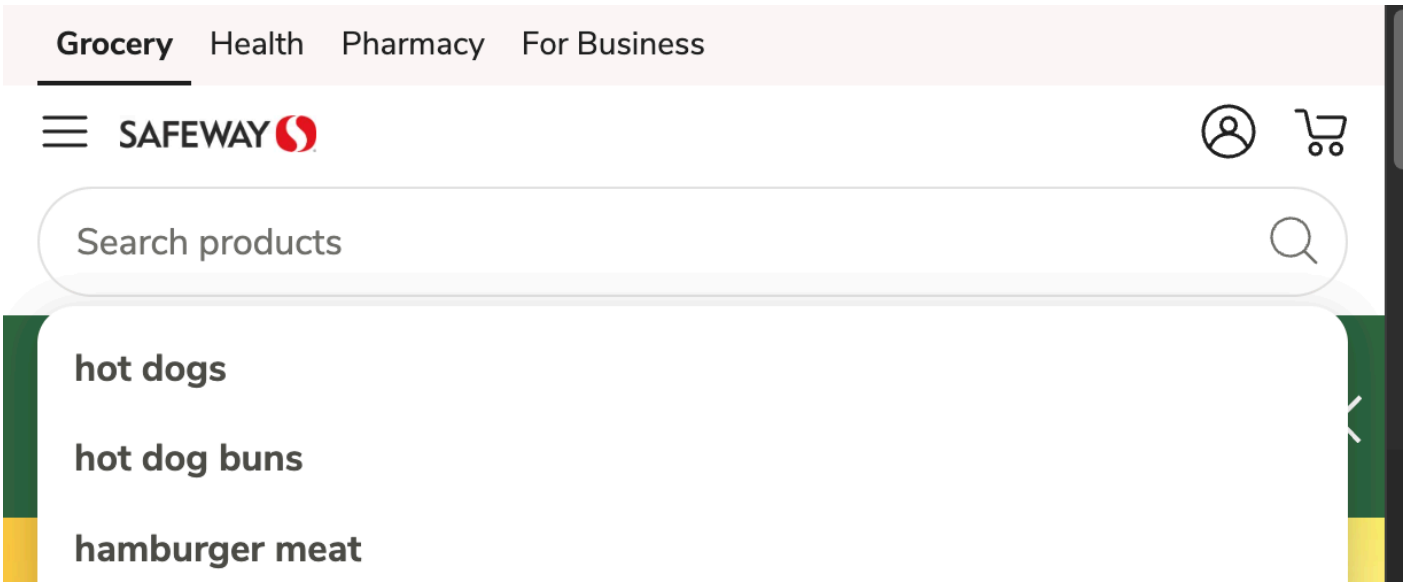
Most websites and applications operate with under the same paradigm.

Model-View-Controller(MVC) The user is displayed a particular view (UI) then based on their inputs this is fed into the control (Backend) which ultimately manipulates a model (Database). The model then passes this information back to the view (sometimes control can be an intermediary here) and we have now gone full circle.



The basic flow of any webpage can be understood with this paradigm.

Example

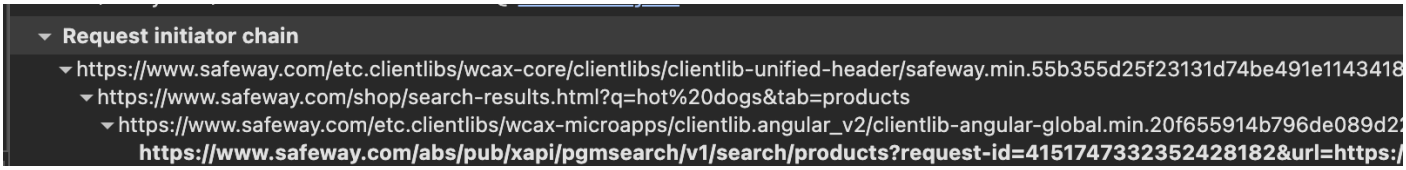


Front end for Safeway contains an input field for searching products. (View)

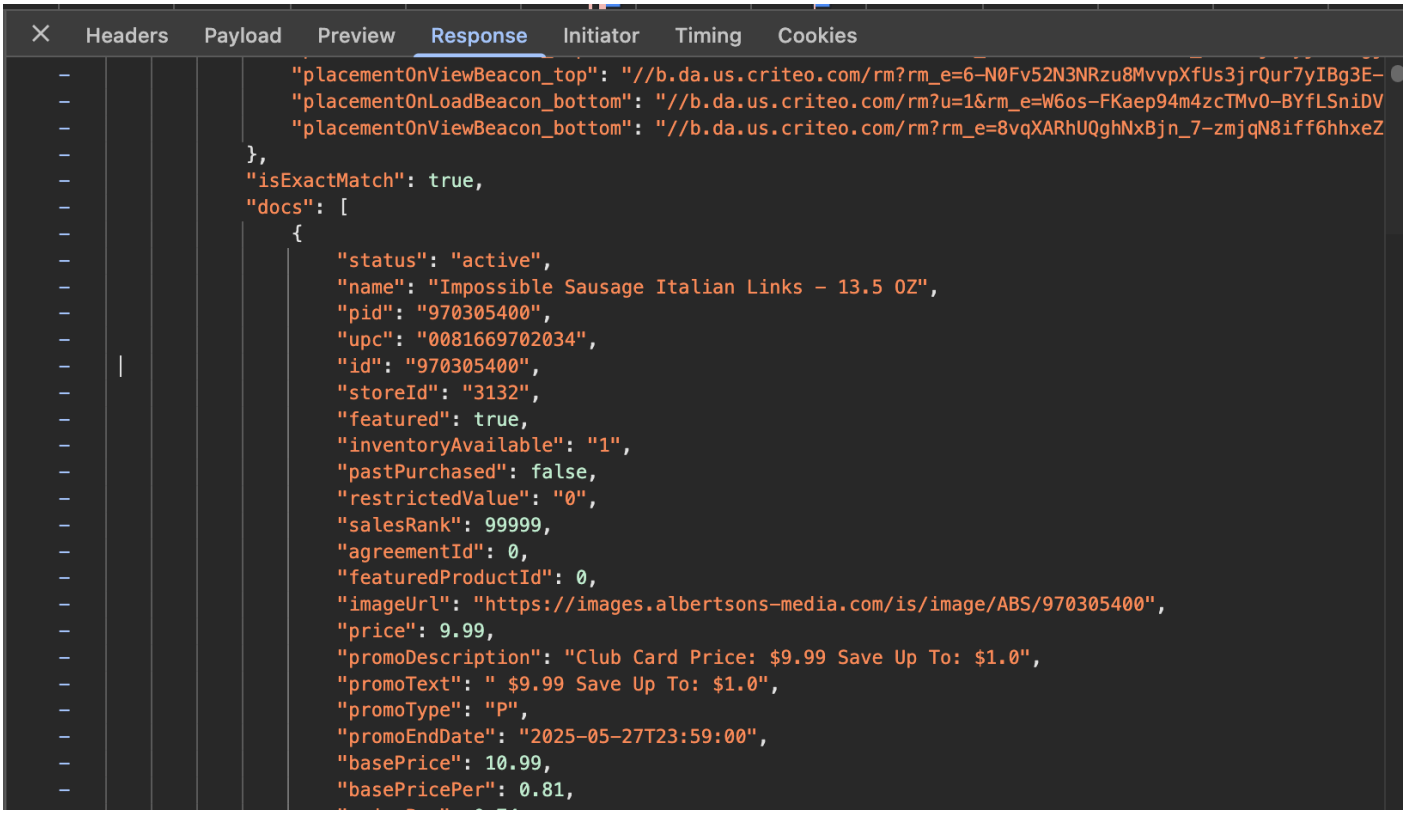
```
<div class="search-wrapper search-nav">
  <form class="search-nav__search-form" action="/shop/search-results.html" role="search" name="search-form" style="padding: 0px;">
    <div class="search-container" data-searchtype="productSearch" data-autosuggestsize="10">
      <div class="search-container-wrapper">
        <input id="skip-main-content" class="search-nav__input form-control input-search ecomm-search product-search-enabled body-m" name="q" autocomplete="off" type="search" aria-label="Search combobox editable has autocomplete search everything at safeway online in store" role="combobox" aria-expanded="true" aria-controls="search-suggestions" aria-autocomplete="list" maxlength="100" placeholder="Search products" data-qa="srch-inpt"> == $0
        <button class="search-nav__icon searchBtn svg-icon-search-grey" data-qa="srch-inpt-cls-btn" aria-label="search">...</button>
        <div class="psAria sr-only" aria-live="polite">
        </div>
        <div class="search-suggestion-container" style="display: block;">...</div>
        <div class="search-suggestion-container-auto"
```

The `<input>` element is inside of a `<form>` element which executes an action when the form is submitted. We can see that the action is `/shop/search-results.html`. This means when a user clicks the icon (or potentially presses enter) that we will send a request to `https://safeway.com/shop/search-results.html` which will likely result in a database call. (Control)

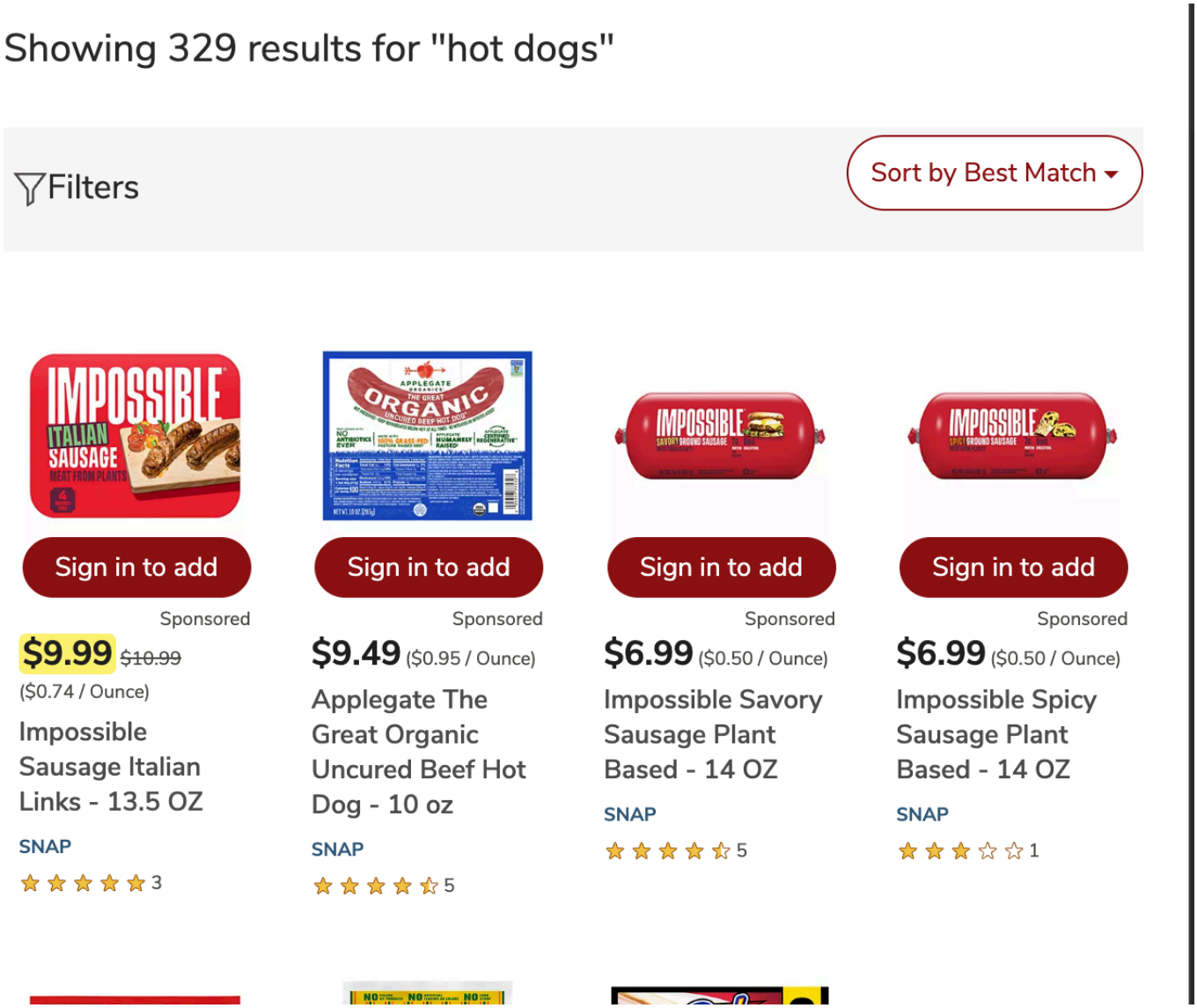
We can further see from this request call chain the flow of this operation:



We go from `https://safeway.com/shop/search-results.html` through a couple of weird likely advertising based requests and finally to [https://www.safeway.com/abs/pub/xapi/pgmsearch/v1/search/products?request-id=4151747332352428182...`](https://www.safeway.com/abs/pub/xapi/pgmsearch/v1/search/products?request-id=4151747332352428182...`)



This image shows that the model has returned the requested information and now when we go back to the view we can see it has populated our UI



Some Helpful Tips/Info

It is typically impossible for us to directly interact with the model. This would imply that the database is public facing (which is extremely unlikely / someone is getting fired behavior).

So as a scraper our scraping ability is dictated by what we can grab from the view or the controller as both of these will be interacting with the underlying model.

Helpful Utilities

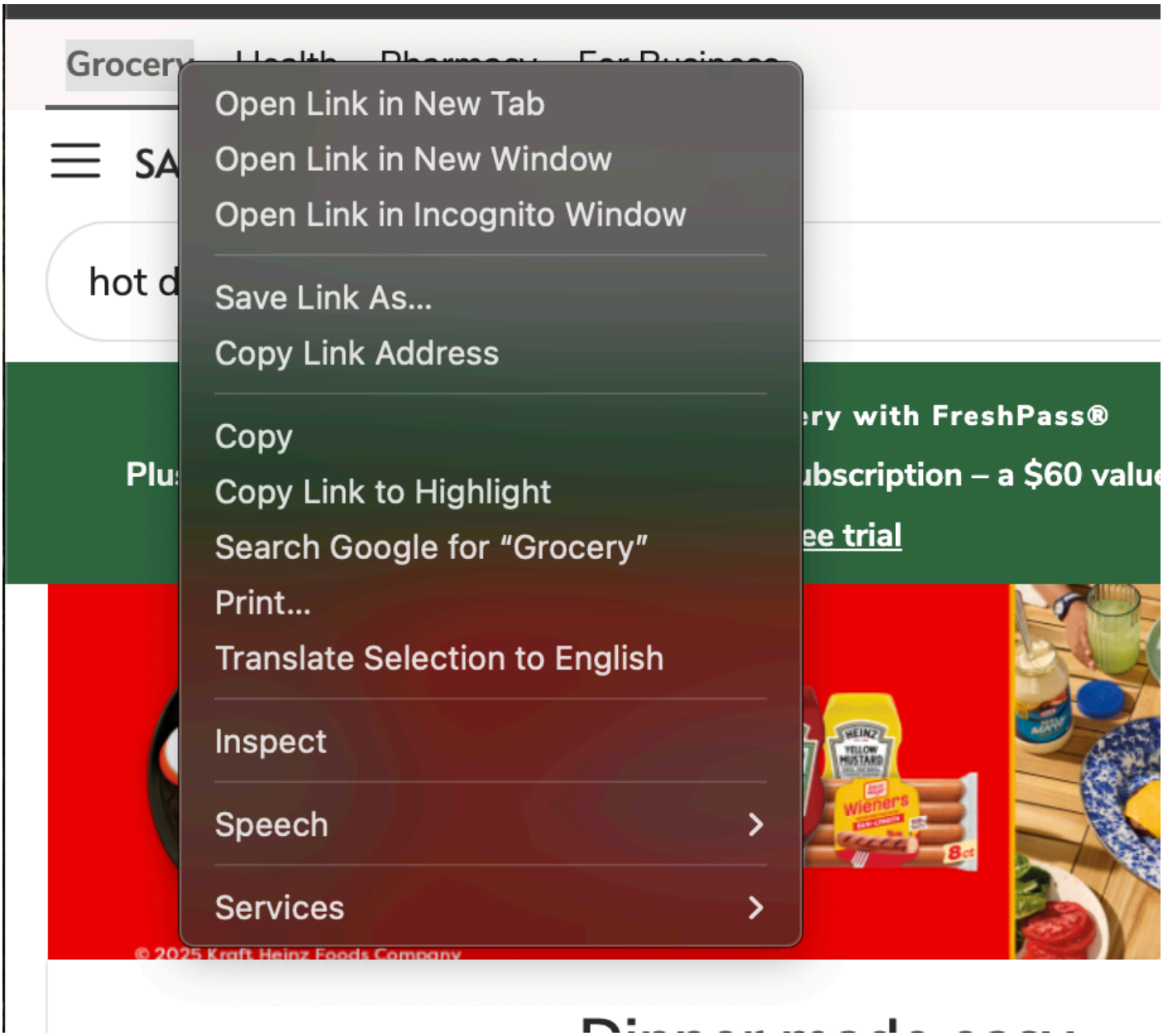
Web scraping involves a lot of trial and error. You will need to spend sometime understanding how an application is behaving before you can fully get the data you want out of it. A lot of your time scraping will be spent discovering how these applications are working with a very small amount of time actually dedicated to scraping. Once you have figured out how to scrape the actual scraping is typically the easy part.

In order to help discover what is going on under the hood there are a couple of useful tools you can use. I will also hopefully cover some useful ways to actually utilize these tools for scraping.

Inspecting Elements (Elements Tab)

When you are on a webpage you can right click almost any element (the caveat is an element called a `<canvas>` which essentially allows drawing. A developer might use this to build dynamic graphics etc..).

Going back to the Safeway example let's say I want to understand what action is performed when I click on one of the navigational links.

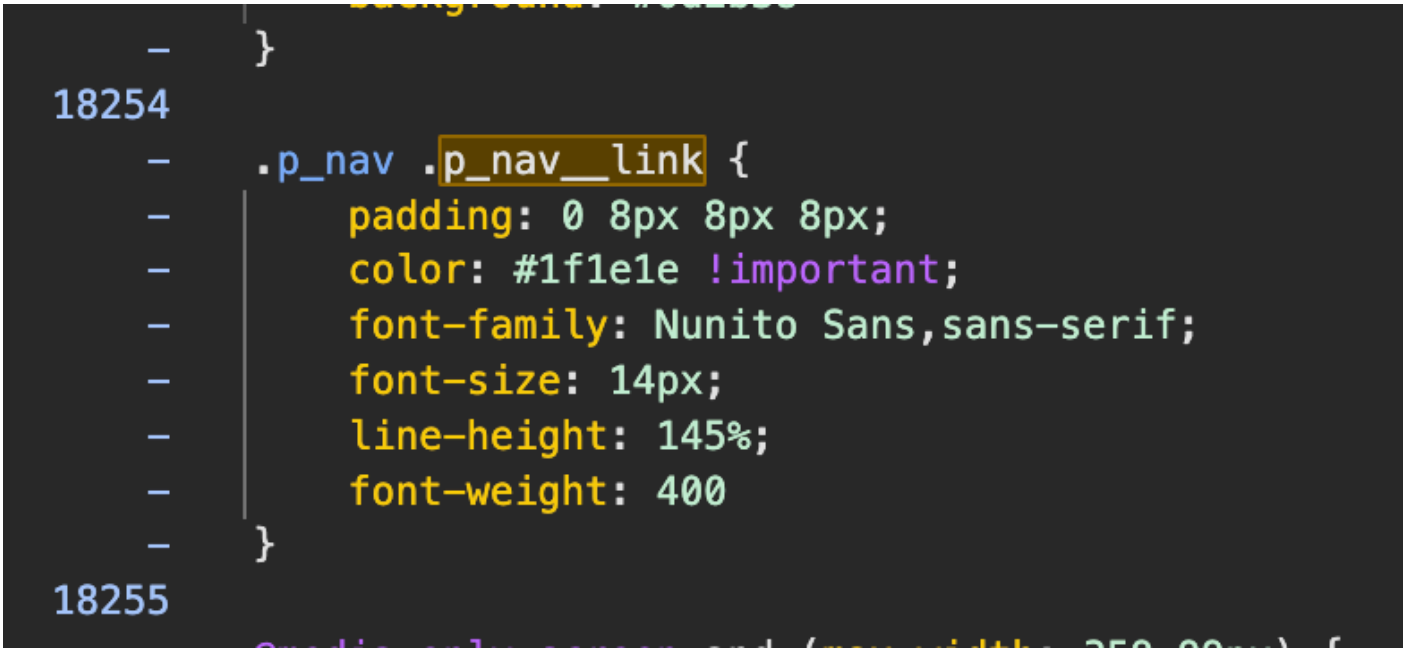


I can now see what happens under the hood when this element is used.

In this instance the 'Grocery' tab might be a bad example because it just leads us back to our current page `safeway.com` but if we look down at the `health` tab we can see it routes a user to `https://safeway.com/health.html`.

We can also see some of the identifying information about the element. The `class` field represents what group of styling elements this element belongs to. Somewhere in a `.css` file there is likely a specific style for `p_nav__link`. This kind of information can be helpful in performing group finds on the html later (if we wanted to `find_all` nav links we could likely use this css group to gather them all).

We can see the css styling applied here



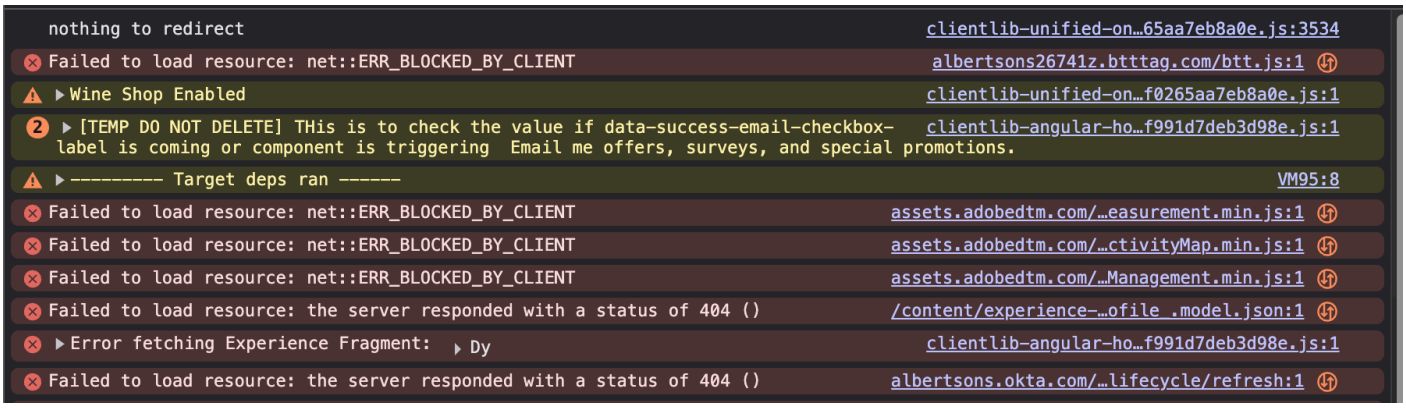
The **Elements** tab is very useful for seeing some general information about the structure of a site. A lot of the times especially with modern webpages however they can be extremely bloated. I will

cover how to deal with parsing these effectively in a later section (Likely the `Requests` section). However this tab is still fundamental to scraping and useful when trying to get started.

The Console Tab

The console tab is almost exactly what it sounds like. This is where print statements from the webpage code will go if there are any. A lot of the times you will see any http error codes in the console if your browser was unable to reach certain resources.

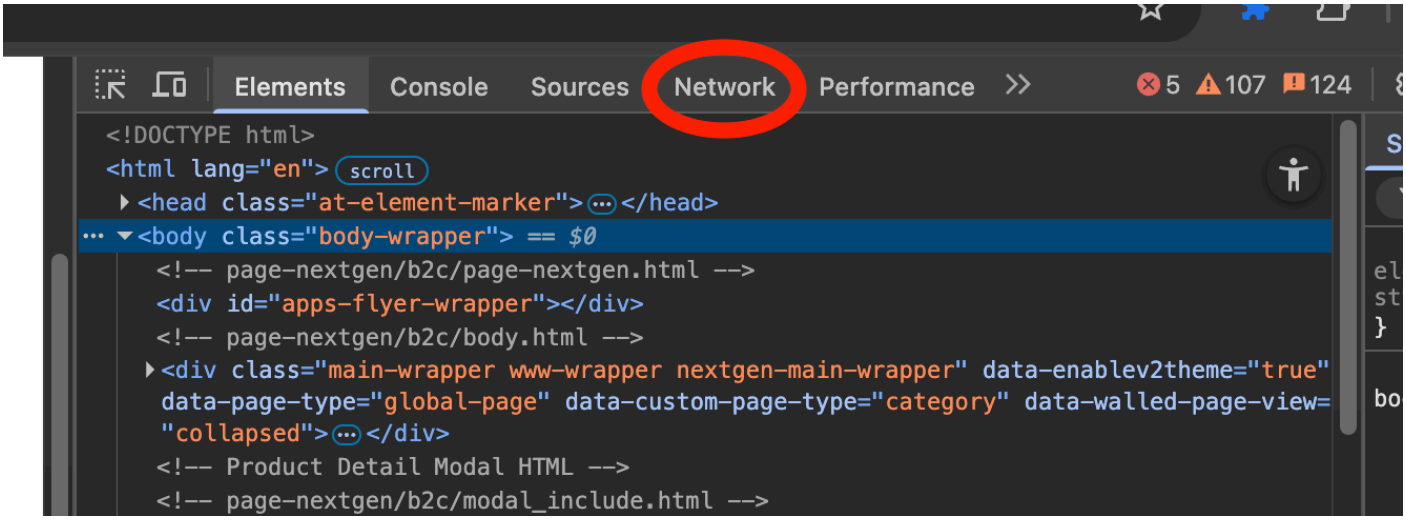
My browser automatically blocks a lot of requests that deal with advertisements so my console is always full of errors. Most websites / browsers will not encounter this problem.



Not the most helpful tab (typically this is helpful for web developers creating an application) but it is still good to know about.

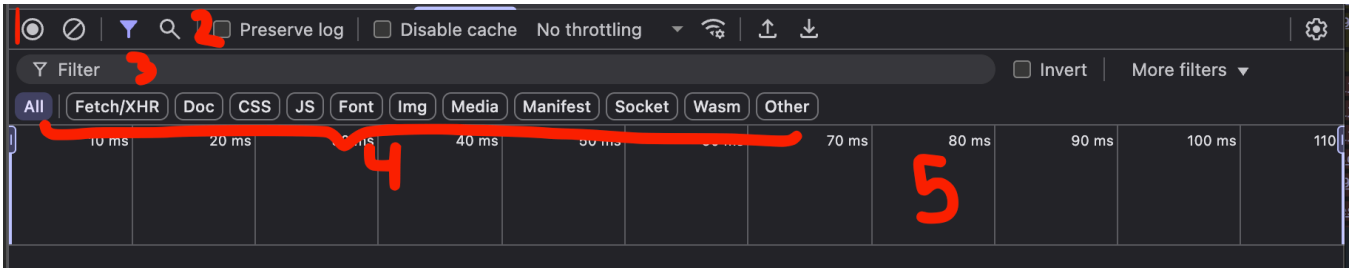
The Network Tab

This has to be one of my favorite debugging tabs ever. Once you understand how the networking tab works you will be able to discover website function at a much deeper level.



There are a lot of useful things within this tab.

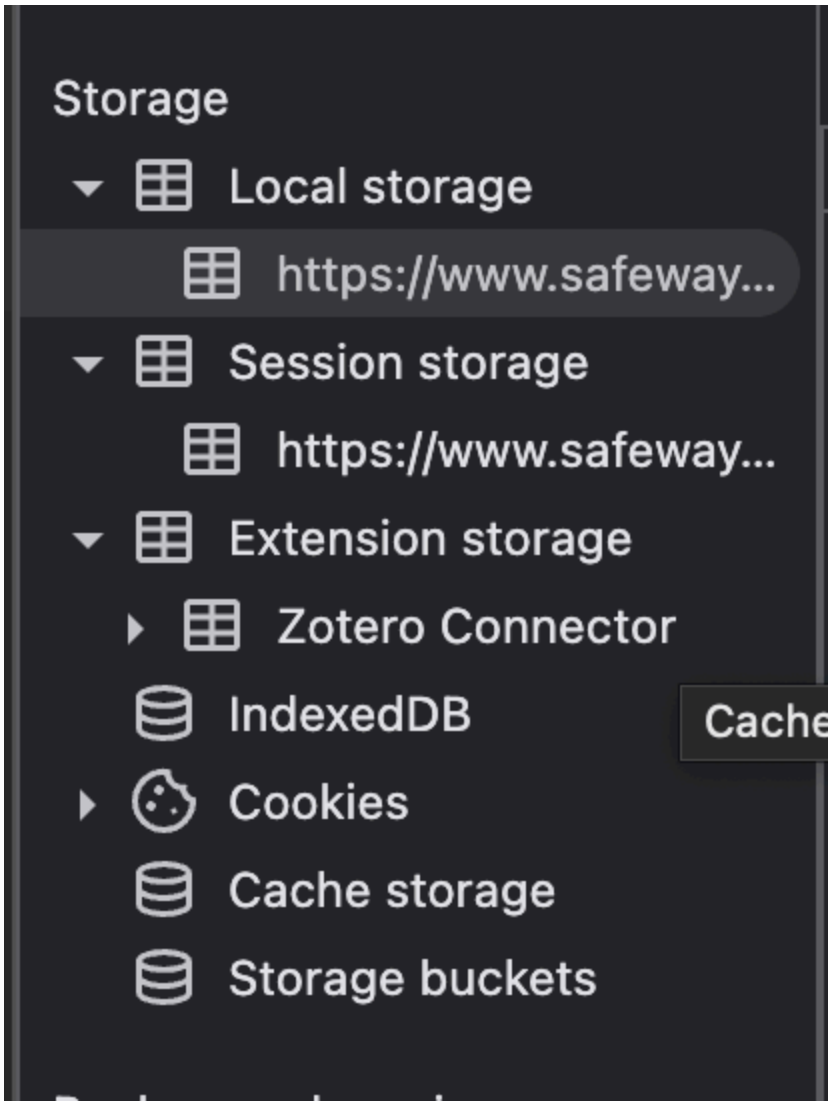
1. The record button. When this button is clicked the icon should turn red. This enables the browser to record outgoing traffic from the page.
2. The preserve log options can maintain the recording across pages (if your page gets moved).
3. The filter box can be used to search for particular requests. If I wanted to see where the `shop/search-products.html` request was made I can just type that into this box.
4. These represent different types of requests. The important ones we care about will mostly be `Fetch/XHR`, or `Doc`.
    1. `Fetch/XHR` represents our browser making a call to an external functional resource like a database call, a lot of the times these are used for advertising information
    2. `Doc` represents actual html documents we are requesting. This will show things that generate `.html` files for our viewing.
5. The timeline. This shows us when calls were made relative to the starting of the recording. This can be useful in situations where you can see some items are loaded in later than the page so that you can further inspect what calls were made that populated the page.



You will likely want to have a filter on like `fetch/xhr` or `doc` because the `all` tab will get populated very quickly with a lot of unnecessary information (at least for scraping)

The Application Tab

This is not the most important tab but I felt it deserved it's own section. This tab keeps track of any stored items in use by the application / your browser.



There are a couple of options here but we primarily will only ever really care about `Local Storage`, `Session Storage`, and `Cookies`.

For example if you sign into an application more often than not your auth token gets stored within `Local Storage`.

Not going to dive too deep into this but this can be helpful if you are facing some errors or issues and want to remove stored information. (I.e maybe you want to trigger the auth flow again so you can delete the token value)

Other Tabs

There are a lot of other tabs, I don't really use them too much but I'm sure they serve their own purposes. Sorry not going to dive into them. If you know of another tab that is beneficial let me know and I can update this guide.

Requests

Let's put the above information and utilities into practice and actually scrape something.

Typically most scraping can be done with enough digging, understanding, and a very small amount of Python dependencies.

Important Scraping Note

During discovery / debugging it can be very easy to make a lot of requests over and over. However you have to be somewhat careful about this in practice. A lot of websites will monitor certain behaviors and start limiting your requests / even outright block you from making them for a while.

A good way to get around this in the early phases of scraping is to **save results you get from your requests locally**. This will allow you to make a single request, view it's structure, and make decisions about what to do next with out sending a request every single time.

e.g

```
import requests as r
req = r.get('https://some_domain.com/endpoint')

with open('some_file.html', 'w') as f: #change the extension depending on what
you are requesting but the idea remains.
    f.write(req.text)
```

Python Libraries

- `requests`: Provides a very simple http/s calling library with some useful abstractions that enable us to make quick one-liner requests to urls/endpoints
- `beautifulsoup4`: Provides an efficient way to navigate html DOM trees. *Documentation is not my favorite*
- `glom`: Provides a nice way to extract data from deeply nested object structures turning something like `obj['field_a']['field_b']['field_c]['field_d]` into `glom(obj, 'field_a.field_b.field_c.field_d)` while also being able to handle default values and errors cleanly. This one is a little more optional but I recently found out about it and felt it was worth a mention.

Example Scraping Task with Requests

An important thing to do before starting to scrape is to have an idea of what you want to scrape.

Im going to break down a couple of examples, and when you would employ certain strategies. We will start with just trying to extract our data from the View and move on to grabbing the information

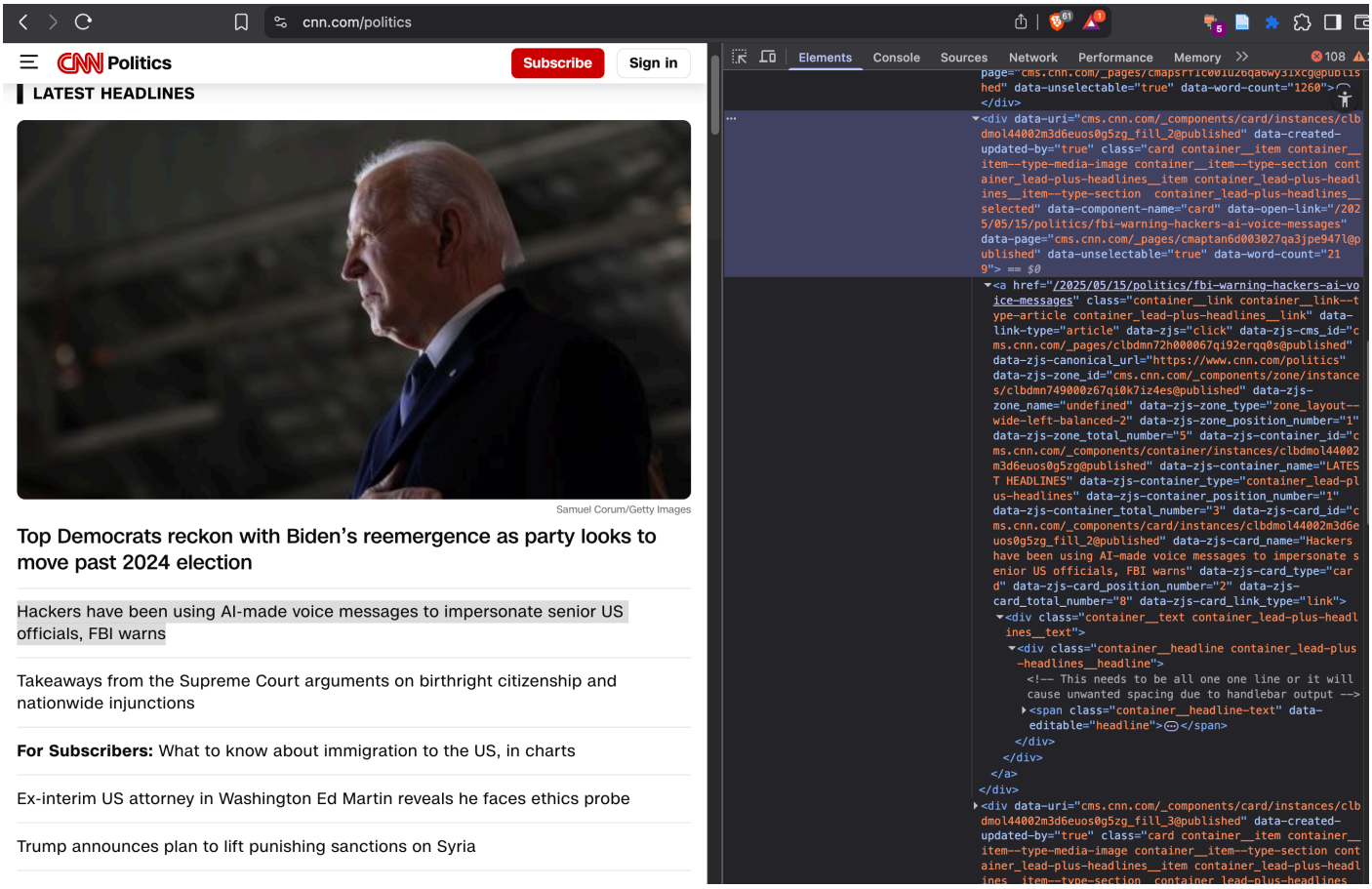
from the Control.

View Scraping (Naive Approach)

Let's install some basic dependencies `pip install requests beautifulsoup4` optionally install `jupyter` to mess around with this in a notebook (highly recommended)

Now for this to work properly we need to find a site that is loading all of the elements we want into the webpage statically instead of dynamically. This is kind of tricky with modern web pages but it can still be a helpful skill to know.

For a simple example we will be scraping (headline, link) pairs from https://www.cnn.com/politics



When we go to this site we can see there is a `div` element containing a couple of other elements. The ones we care about are the `<a>` tag, and the `<span>`. We can see that the `<a>` tag contains the link to the article, and the `<span>` tag contains the headline text.

Make the request

```
import requests as r

url = 'https://www.cnn.com/politics'
```

```
cnn_req = r.get(url, timeout=2)

cnn_req.raise_for_status()
```

Let's set up a variable for the URL and pass it in the `get` request we are making with the `requests` library.

The `timeout` variable gives us an upper bound time limit on the request in case something goes wrong we don't want to continue trying to connect infinitely. You don't always have to specify this variable but it can be good practice.

finally we call `cnn_req.raise_for_status()` this will throw an error to our interpreter if we get a HTTP code in the `400-599` range indicating that the request was not successful. Again you don't have to do this but it can be a good way to ensure that you have the request data before operating on it. If you don't use this you will likely need to establish some conditional statements to ensure the request was responded to.

Parse the DOM tree

```
from bs4 import BeautifulSoup
html = cnn_req.text

soup = BeautifulSoup(html)
```

Here we are just storing the html data from the request into a variable and passing it into the BS4 parser to easily navigate the DOM tree.

Navigate the DOM tree

```
data = soup.find_all('div', 'card')

items = [(item.span.get_text(), item.a['href']) for item in data]
```

Firstly we use our soup variable from earlier to quickly find the elements we are interested in. The first class specified on on this element in the inspect tab is `card` so I started there. Sometimes you will need to debug this phase because multiple classes can be used on an element. You will often need to discover the correct one and make sure it works.

The second thing I do here is a bit of shorthand but I can explain each part.

We are iterating over each of the found `card` class elements found in the soup. We can then navigate the elements locally by specifying the child tags.

We know there is a `span` within this `div` that contains the headline information we want. So we can call `item.span` to quickly get the child `span` element of our found item. We then call `get_text()` to only grab the actual text information within that element.

We also know there is a `a` tag within the `div` so we do the same thing with `item.a` to get that element. We then access the `href` property much like accessing a field in a dictionary via `a['href']`

Final Code

```
import requests as r
from bs4 import BeautifulSoup

def get_headlines()->list[tuple]:
    """Get Headlines, Link Pairs from CNN"""
    url = 'https://www.cnn.com/politics'
    cnn_req = r.get(url, timeout=2) # Set up GET request to our target URL
    cnn_req.raise_for_status() # Raise an error if we don't get a 200

    html = cnn_req.text # Get the HTML from the response
    soup = BeautifulSoup(html) # Parse HTML into a easily navigable DOM tree
    data = soup.find_all('div', 'card') # Find the elements that contain the
info we want

    items = [(item.span.get_text(), item.a['href']) for item in data] #grab the
data from each element

return items

import json #just saving the results to a file. you don't have to do this
with open('some_file.json', 'w') as f:
    json.dump(get_headlines(), f, indent=4)
```

This is a pretty simple example but feel free to experiment here now that you have some of the basics down.

For something similar try and scrape the headline, link pairs on this page https://apnews.com/politics. Specifically you can look at the **More Politics News** sections. You will not have to re-write the entire code and should be able to plug in play with a couple of the variables we have already established to accomplish this task. You could even potentially figure out a way to make a couple of abstractions that would make this code work for both **CNN** and **APNews**.

Control Scraping

One of the main problems with the **View Scraping** technique is that we are bounded by what is actually loaded onto the `.html` that is returned to us. If only two elements are populated at request time we will only get those two elements. There is also a large amount of information returned because we are receiving all of the html data like style sheets, and elements we do not even care about.

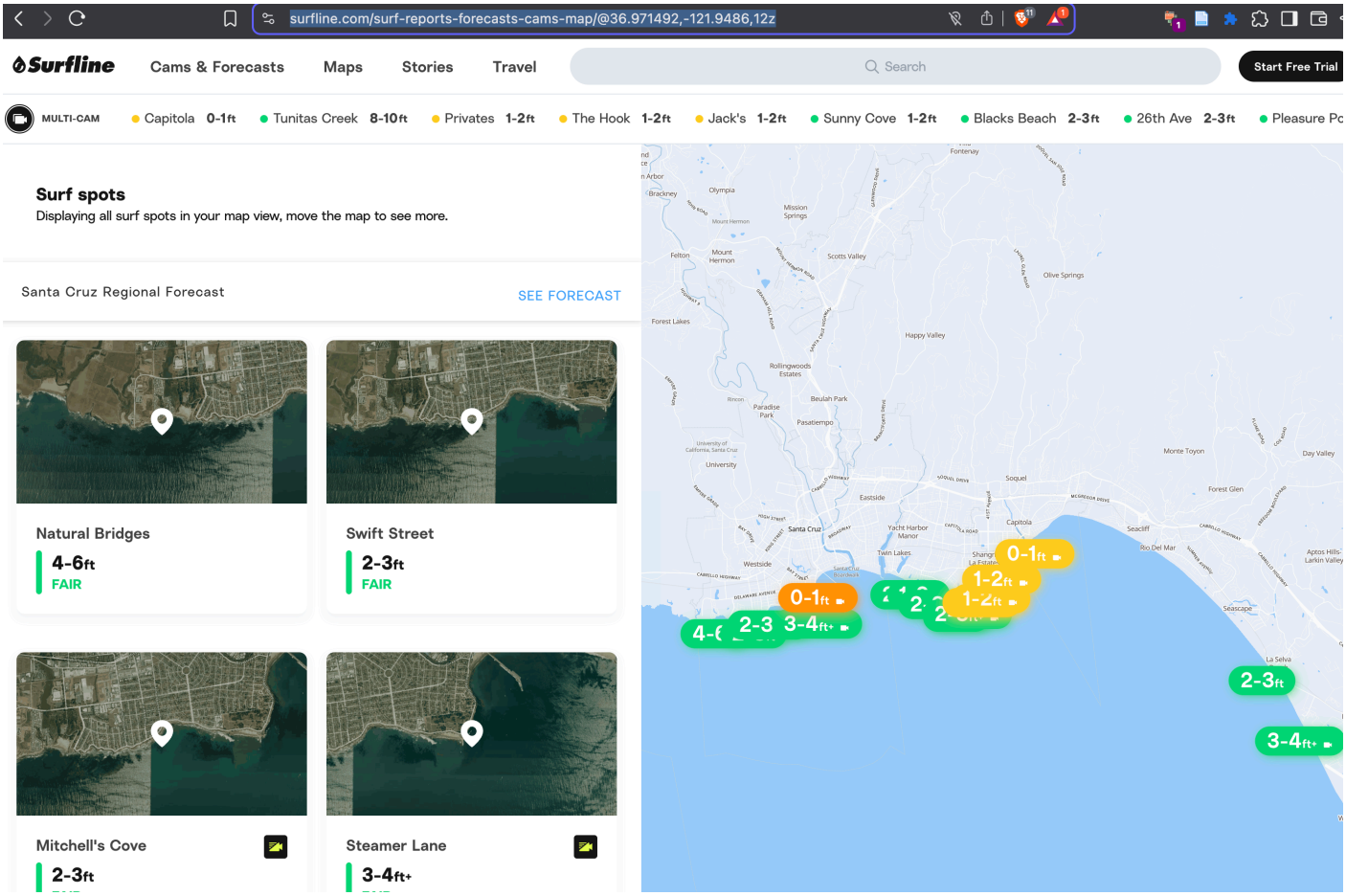
However using what we know about the **MVC** paradigm we can often interact with the backends ourselves. Typically allowing us to efficiently grab the data we are looking for. Most of these endpoints will return a nice JSON object further enabling us to discover more information about certain items.

For this approach will need `requests` (and optionally `glom`)

Let's take a look at https://surfline.com and try and take some spot information.

Discovery

If you open up the "Maps" tab you will be redirected to something like `https://www.surfline.com/surf-reports-forecasts-cams-map/@36.971492,-121.9486,12z`
We can see that this URL contains parameters representing latitude, longitude, and a zoom level.



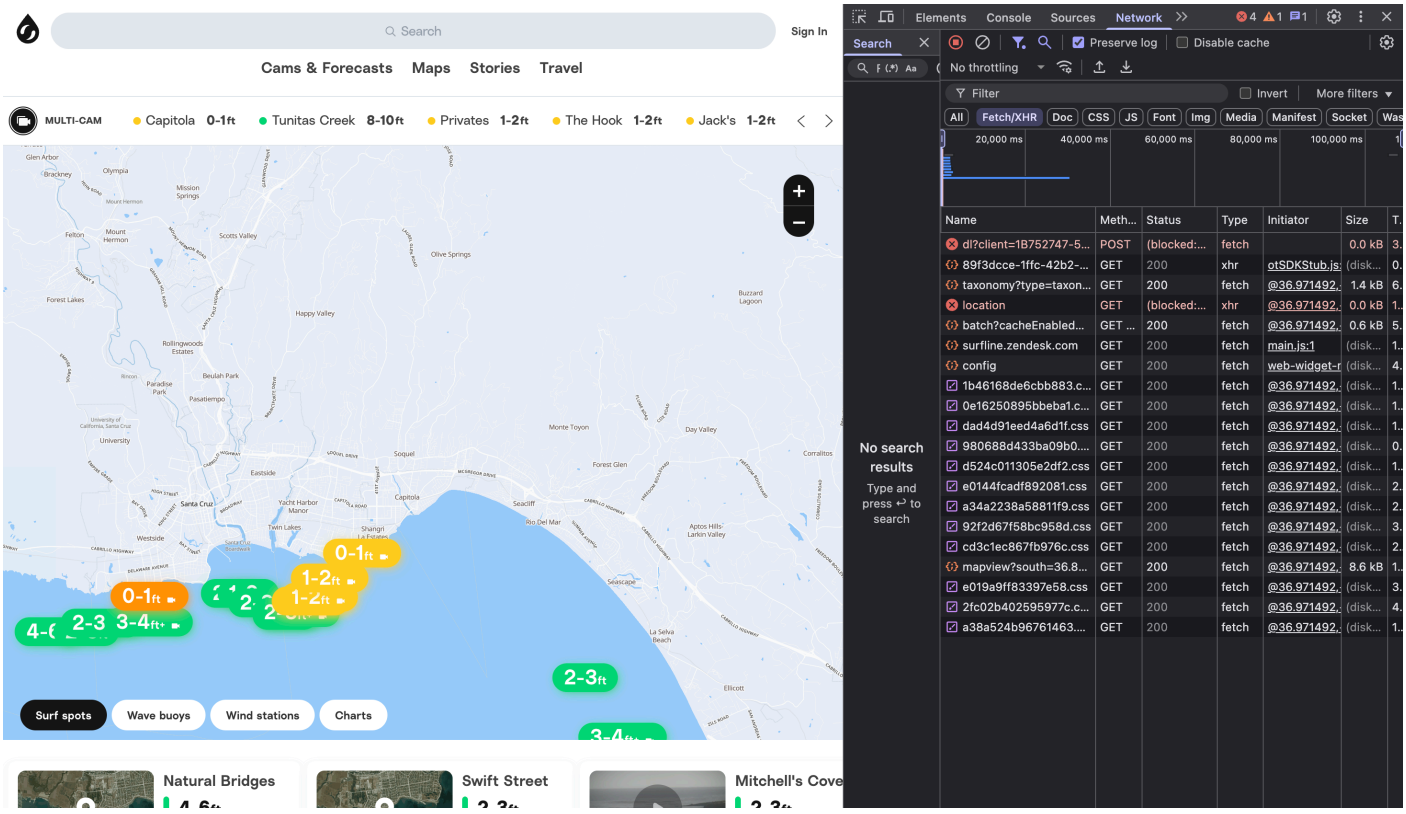
When we try to just get the webpage we get a **403: Forbidden** error.

```
surf = 'https://www.surfline.com/surf-reports-forecasts-cams-map/@36.971492,-121.94

surf_req = r.get(surf)

surf_req.raise_for_status()
```

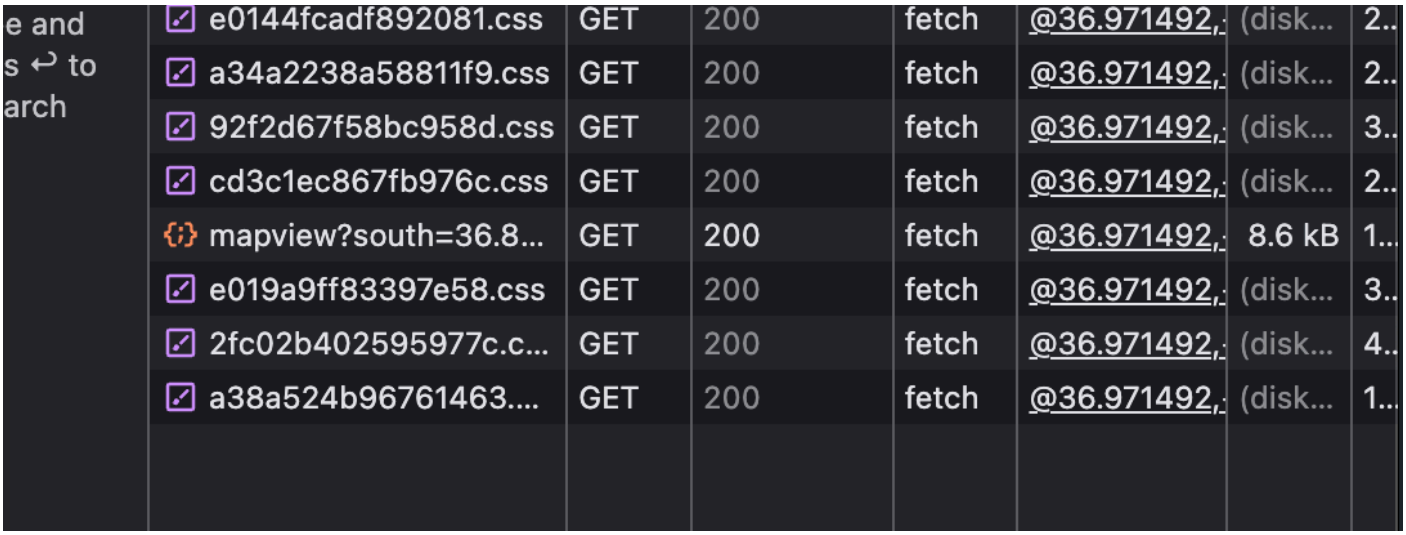
⊗ 0.2s Python

```
---------------------------------------------------------------------------
HTTPError                                 Traceback (most recent call last)
Cell In[2], line 5
      1 surf = 'https://www.surfline.com/surf-reports-forecasts-cams-map/@36.971492,-12
      3 surf_req = r.get(surf)
----> 5 surf_req.raise_for_status()

File ~/dev/misc/ScrapingTutorial/.venv/lib/python3.11/site-packages/requests/models.py:
   1019     http_error_msg = (
   1020         f"{self.status_code} Server Error: {reason} for url: {self.url}"
   1021     )
   1023 if http_error_msg:
-> 1024     raise HTTPError(http_error_msg, response=self)

HTTPError: 403 Client Error: Forbidden for url: https://www.surfline.com/surf-reports-f
```

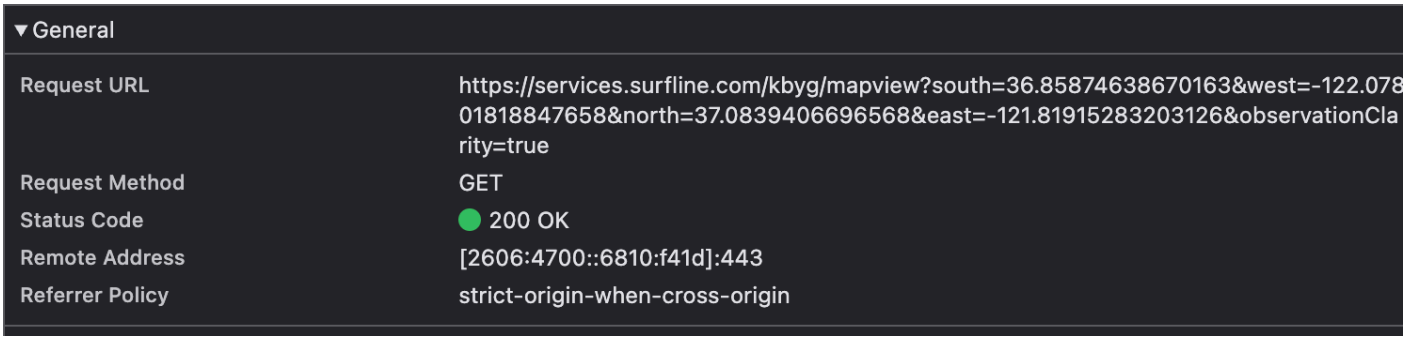
Telling us that View based scraping will not get the job done here. Or at least might require some level of authorization to accomplish.



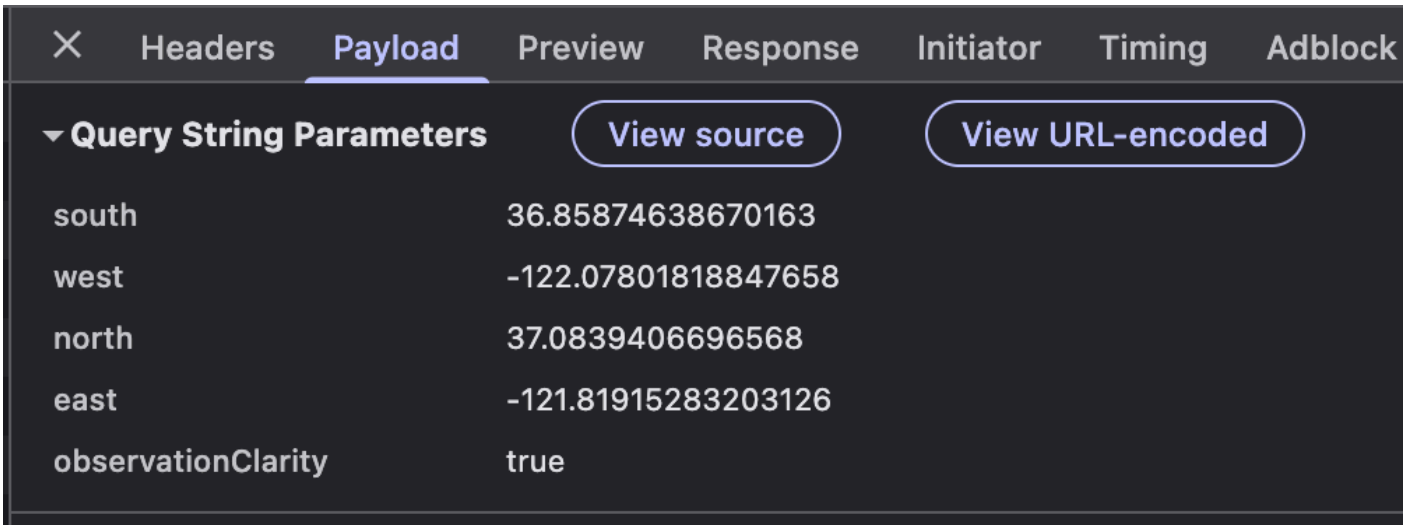
What we can do however is check out the network tab and look for any potential requests that might be carrying the data that we want. Turn recording on and hit refresh on the page to generate a list of requests that occur. Filter for `Fetch/XHR`. Optionally end the recording so that you don't get any further requests cluttering the screen.



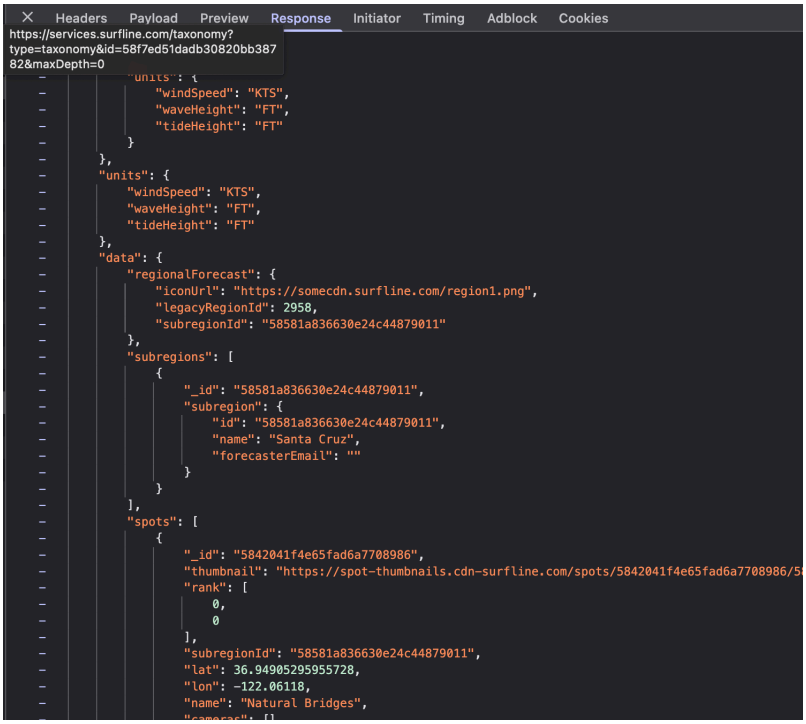
You may have to inspect a couple of requests before you are sure which one to grab. I see an interesting one here called `mapview?south=....`



Looking at this request in more detail we can check out the `Headers` tab. We can see that this was a GET request to the above endpoint.



Looking at the Payload tab we can see the information that was sent. Importantly because this was a GET request the payload is not sent in the body, but it is encoded into the URL we sending.



Finally in the `Response` tab we can see that this request returned a JSON containing very detailed spot information from the map view that was loaded on the browser.

One thing to note here as well is that the `lat`, and `lon` variables for the spots returned in this request are within the bounds specified as [`south`, `north`] for `lat` and [`west`, `east`] for `lon`.

Let's make a request

We saw from the `Headers` tab that our request was sent to `https://services.surfline.com/kbyg/mapview`. So let's set that up.

For now let's just use the parameters we sent through the browser to see if we can get the same response. Later we will explore changing these to gather more / other information.

A very simple approach to doing this would be just flat out requesting the url

```
import requests as r

url = 'https://services.surfline.com/kbyg/mapview?
south=36.85874638670163&west=-122.07801818847658&north=37.0839406696568&east=-1
21.81915283203126&observationClarity=true
```

However this URL is extremely large due to the parameters being encoded within it.

One potential way to handle this could be to use an `f-String` and place our information in it like so

```
url = f'https://services.surfline.com/kbyg/mapview?south={south}&west=
{west}&north={north}&east={east}&observationClarity={observationClarity}'
```

This will lead to it's own problems. Maybe not in this particular request but by directly placing the information into the string this way we could potentially be sending information incorrectly to the site. Certain information needs to be encoded in particular ways to be usable i.e space characters are encoded as `%20` in a URL and so on. We also would have to manually add in any other arguments we want to use and make sure we separate them by an `&` token.

There is a much simpler, and easier way to accomplish this using the `requests` module

```
import requests as r

base_url = 'https://services.surfline.com/kbyg/mapview'

params = {
    'south': 36.85874638670163,
    'west': -122.07801818847658,
    'north': 37.0839406696568,
    'east': -121.81915283203126,
    'observationClarity': True
}

spot_req = r.get(base_url, params=params) #by setting the params
# the requests module will automatically encode our query into the URL
spot_req.raise_for_status() #always good to have this
```

When this code is run you can inspect the `spot_req.url` and will see `'https://services.surfline.com/kbyg/mapview?south=36.85874638670163&west=-122.07801818847658&north=37.0839406696568&east=-121.81915283203126&observationClarity=True'` as it's value.

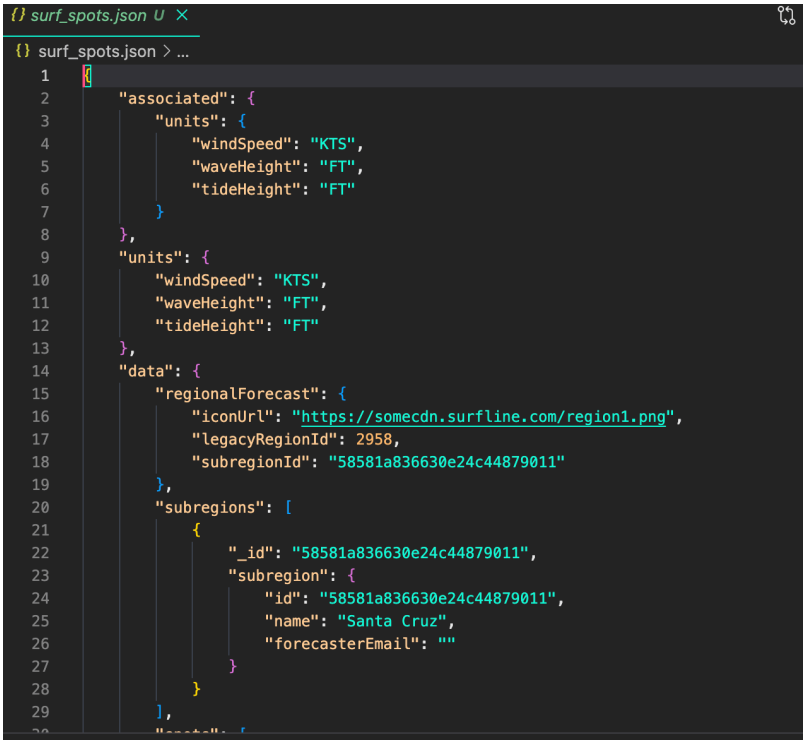
Getting the data out

To finally get all of the data out all we need to do is call

```
spot_req.json()
```

So our final code should look something like this

```python
import json
import requests as r

base_url = 'https://services.surfline.com/kbyg/mapview'

params = {
'south': 36.85874638670163,
'west': -122.07801818847658,
'north': 37.0839406696568,
'east': -121.81915283203126,
'observationClarity': True
}

surf_req = r.get(base_url, params=params)

surf_req.raise_for_status()

with open('result.json', 'w') as f: #store results to a file
    json.dump(surf_req.json(), f, indent=4)
```



You should see something like this depending on the arguments you sent in the request.

Optional `glom` parsing

Glom allows us to easily navigate nested objects. Instead of needing to pass in something like `spot_data['data']['spots]` we can easily grab the spot items with `glom(spot_data,`

`'data.spots')`. The first argument represents the `target` object. The second argument represents the `spec` we are searching for. We can optionally pass in multiple specs

```
# ... code placing the json into an object called spot_data

from glom import glom

specs = {
    'spots': 'data.spots',
    'spot_names': ('data.spots', ['name'])
}

parsed = glom(spot_data, specs)
```

This will make two objects within our parsed object containing a list of all of the spots and their associated information. As well as a simple list of all the spot names. The former is too big / annoying to screenshot, but here is the screenshot of the `spot_names`

```
parsed = glom(j, specs)
parsed['spot_names']
```

✓ 0.0s

```
['Natural Bridges',
 'Swift Street',
 "Mitchell's Cove",
 'Steamer Lane',
 'Cowells',
 'Blacks Beach',
 'Sunny Cove',
 '26th Ave',
 'Wind and Sea',
 'Rockview',
 'Pleasure Point',
 "Jack's",
 'The Hook',
 'Privates',
 'Capitola',
 'La Selva',
 'Manresa']
```

Putting it all together (Our own Surfline API)

Let's take everything we have learned in this section to build a simple tool that can scrape this spot information for us.

```
"""Control based scraping / Endpoint scraping"""
import json
```

```
import argparse
import time

import requests as r
from glom import glom

BASE_URL = 'https://services.surfline.com/kbyg/mapview'
TIMEOUT = 5

def get_spot_data(spot_params):
"""Get spot information based on query parameters"""
    req = r.get(BASE_URL, params=spot_params, timeout=TIMEOUT)
    req.raise_for_status()
    data = req.json()
    out = {
        'params': spot_params, #sending our params to correlate info
        'data': glom(data, 'data.spots') #grabbing just the spot data
    }

return out

def main():
"""main func"""
parser = argparse.ArgumentParser(
    prog='Surfline Spot Scraper',
    description='Gather information on surf spots by making requests to the
underlying api of surfline',
)
# arguments to make this a semi-flexible callable script
parser.add_argument('-p', '--params', default='params.json', help='File for
parameter data')
parser.add_argument('-t', '--time', default=0, help='Time between requests')
parser.add_argument('-o', '--output', default='output.json', help='File to
place output data')

args = parser.parse_args()

params_file = args.params

with open(params_file, 'r', encoding='utf8') as f: #load our saved parameters
    params = json.loads(f.read())

out = []
for param in params: #collect the data on the defined map locations
    try:
        data = get_spot_data(param)
```

```
        out.append(data)
        time.sleep(int(args.time)) #sleep to not overload requests (default 0
aka no sleep)
    except Exception as e: #handle potentially getting an error from the
request
        print(f'ERROR: Encountered {e} while using {params}')

with open(args.output, 'w', encoding='utf8') as f: #save this information
    json.dump(out, f, indent=4)

print('Done Scraping Surfline!')

if __name__ == "__main__":
    main()
```

A quick run down of the above code:

1. We set our request parameters in a file called `params.json` however this is an argument to the script so you can always pass a new/different file
2. Define a timeout (optional), and an output file to manage the time between our requests, and where our data goes
3. Set up a simple function to make our requests based on our passed parameters
4. Load all of our parameters, iterate over them, collect the data
5. Store that data into a file specified by `--output` default is `output.json`

Selenium

Selenium is a library that enables us to control a browser. This is primarily used for web development by automating front end actions. You can have it click buttons, enter information into fields and a lot of other very useful things.

In terms of scraping I view Selenium as a worse case option. You will find in your scraping adventures that many companies / websites do not want to have their data scraped. There will be encrypted auth tokens, request monitoring, etc... that will prevent you from grabbing the data how I laid out in the **Requests** section.

Often times backend APIs can be protected, and the frontend can still be dynamic making it seem almost impossible to utilize standard http/s requests to access this information.

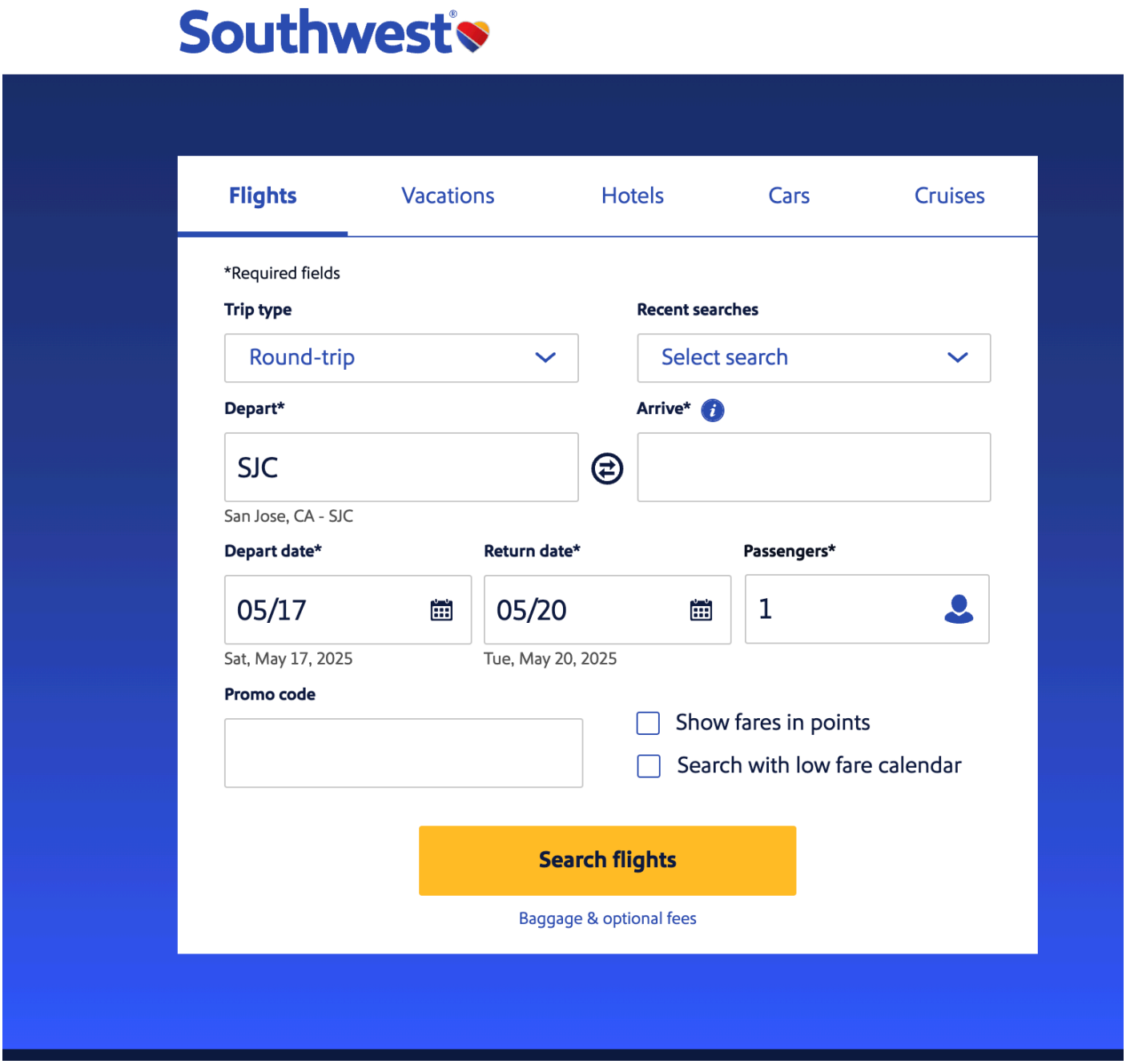
This is where Selenium can come in to help us. By automating the browser's behavior we can essentially act like a user to navigate a website. Then parse the loaded html through `beautifulsoup4` or some other html parser and grab the data that way.

In my experience it is better to exhaust the `requests` library first as the overhead of Selenium is very high.

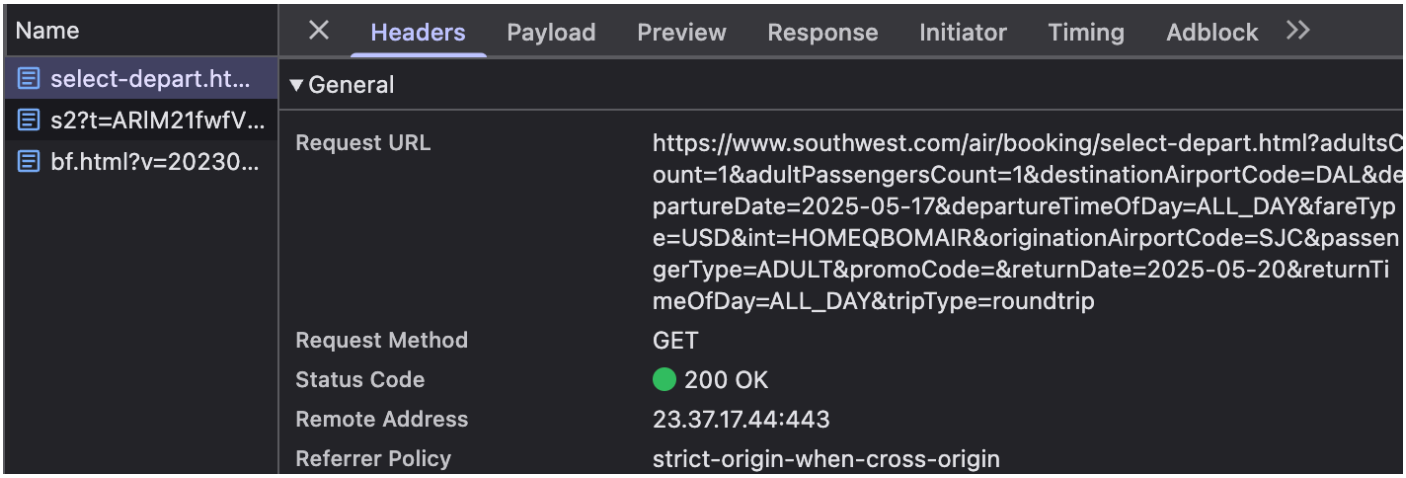
SouthWest Airlines

In making this tutorial I thought it would be fun to scrape ticket information from https://southwest.com. I initially was going to use it for the **View Scraping** section in requests but noticed it would not work for that. I looked at their network calls and thought it might be able to work for the **Control Scraping** but I will show you the problems with that.

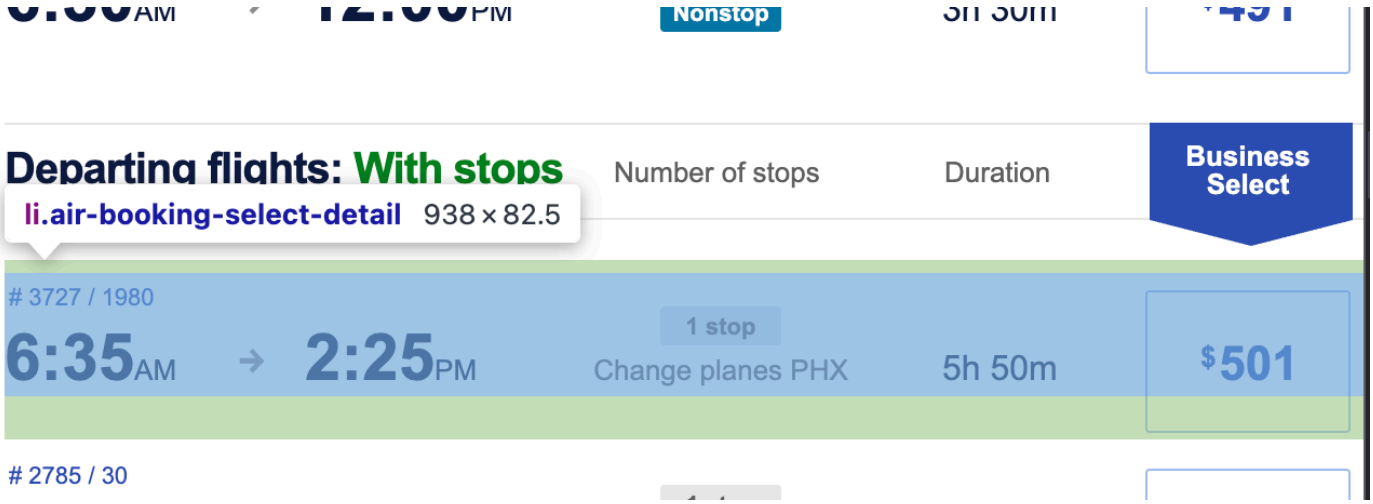
View Based



They have a little form on their front page that let's you input your information to look for flights.



After filling out the form we can see that a document is loaded. We can see that the input fields were placed into the URL as query parameters.



We can see that each flight row is defined by a class `air-booking-select-detail`

```
south_west = r.get('https://www.southwest.com/air/booking/select-depart.html?adultsCo
south_west.raise_for_status()
soup = BeautifulSoup(south_west.text)
soup.find_all('li', 'air-booking-select-detail')
```

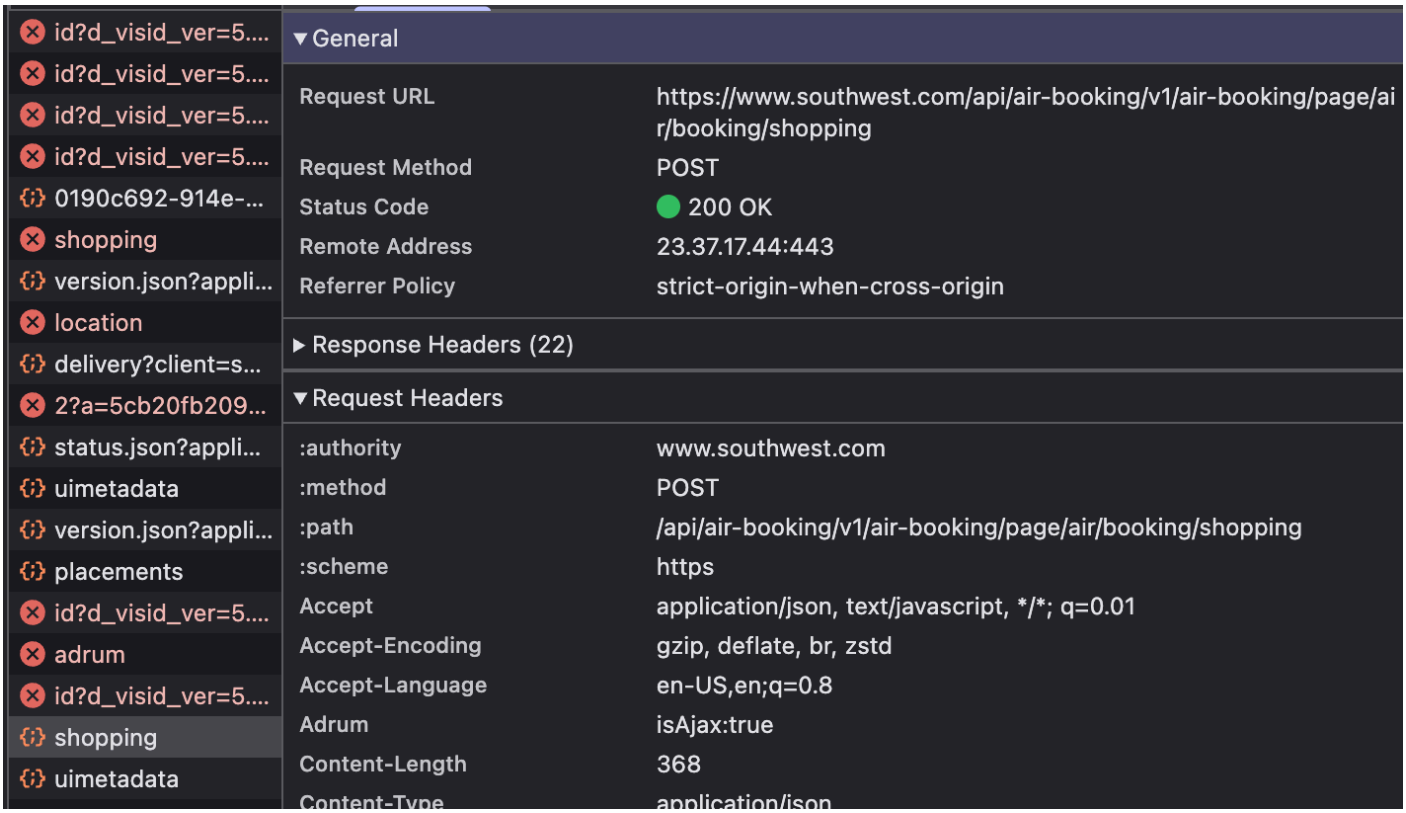
```
7]  ✓ 0.1s  Python
[]
```

However when parsing the DOM tree in python even after a successful we can see that our parser is unable to find the desired elements.

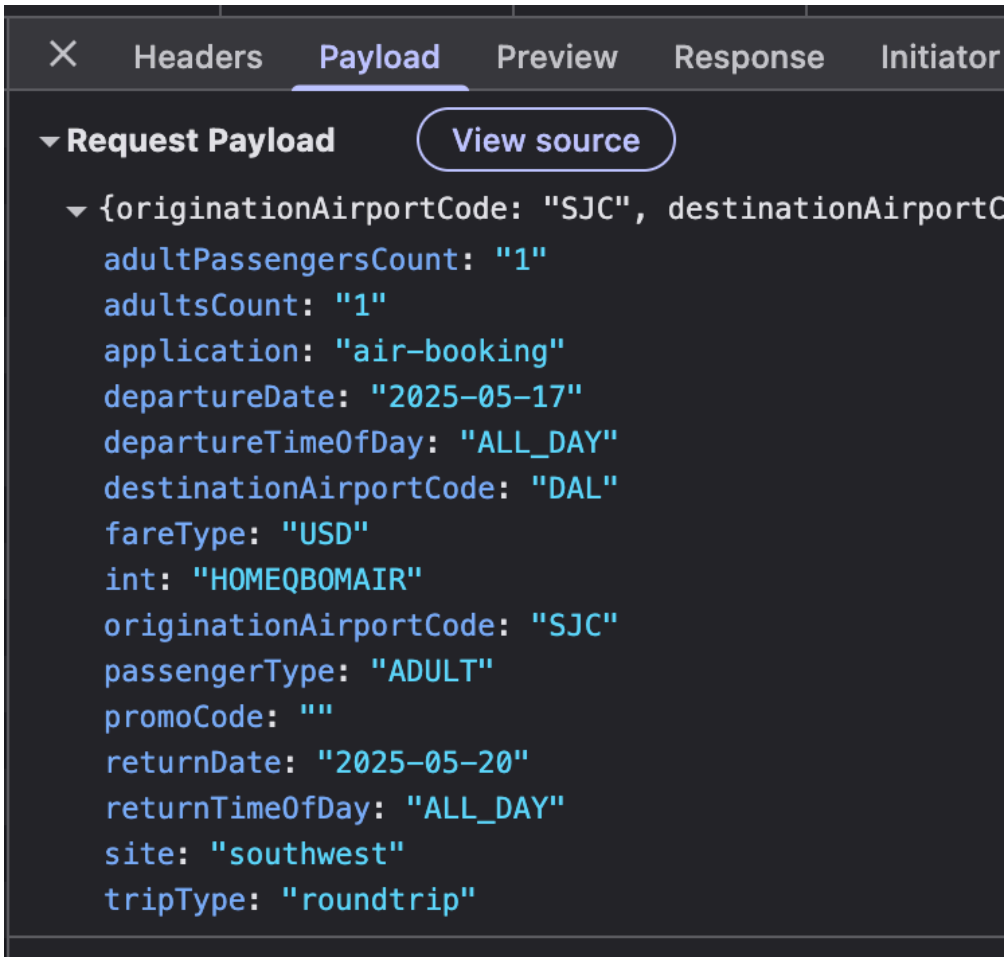
So our View based approach is not going to work here.

Control Based

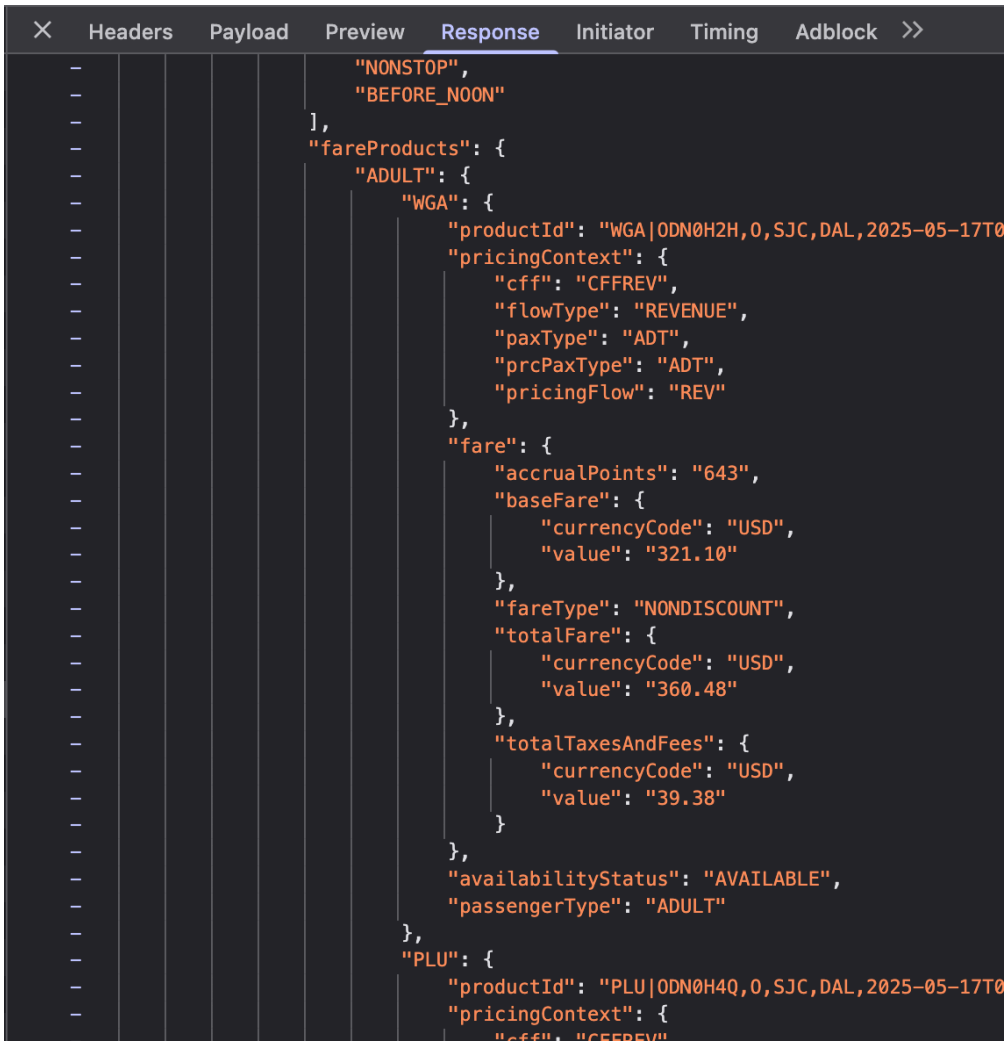
Inspecting the Network tab after clicking the button we see a request to a endpoint labeled `shopping`



It is a POST request and seems to be related to the information we passed in the form.



We can see that the payload is essentially the same as what we input into the form element.



We can also see that the response is a nice JSON format with the ticket information.

```
    south_west = r.post(url, json=payload)
    south_west.raise_for_status()
    south_west.json()
    ⌘L to chat, ⌘K to generate
⊗ 0.1s

---------------------------------------------------------------------------
HTTPError                                 Traceback (most recent call last)
Cell In[19], line 22
      3 payload = {
      4   "originationAirportCode": "SJC",
      5   "destinationAirportCode": "DAL",
   (...)     18   "site": "southwest"
     19 }
     21 south_west = r.post(url, json=payload)
---> 22 south_west.raise_for_status()
     23 south_west.json()

File ~/dev/misc/ScrapingTutorial/.venv/lib/python3.11/site-packages/requests/models.py:1024, in Response.raise_for_status(self)
   1019     http_error_msg = (
   1020         f"{self.status_code} Server Error: {reason} for url: {self.url}"
   1021     )
   1023 if http_error_msg:
-> 1024     raise HTTPError(http_error_msg, response=self)

HTTPError: 403 Client Error: Forbidden for url: https://www.southwest.com/api/air-booking/v1/air-booking/page/air/booking/shopping
```

When making the request with the exact same parameters and information we can see that we are forbidden from making a request to this endpoint. So we will need to re-evaluate our strategy.

Selenium Based

Let's install selenium via `pip install selenium`

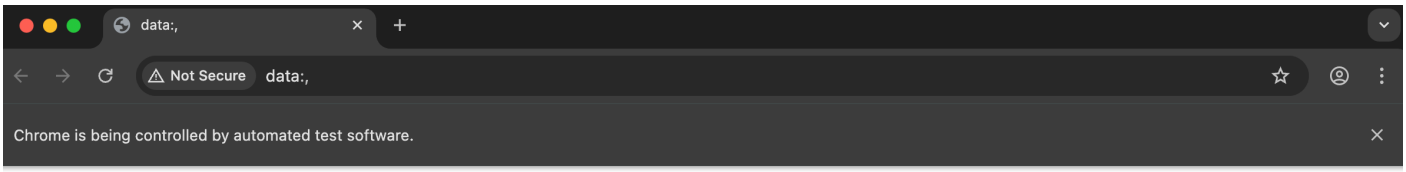
You will also need a driver (aka a browser) for selenium to operate. Use this link and download the right one for you https://selenium-python.readthedocs.io/installation.html#drivers. I typically will use chrome because it seems to have the most stability / most websites re

In the **View Based** section we saw a url `https://southwest.com/air/booking/select-depart.html?<params>`. We were unable to get any helpful data out of that link with just the standard request library. But what would happen if we use that link in Selenium?

Making a driver

To start out we will need to instantiate a webdriver from Selenium

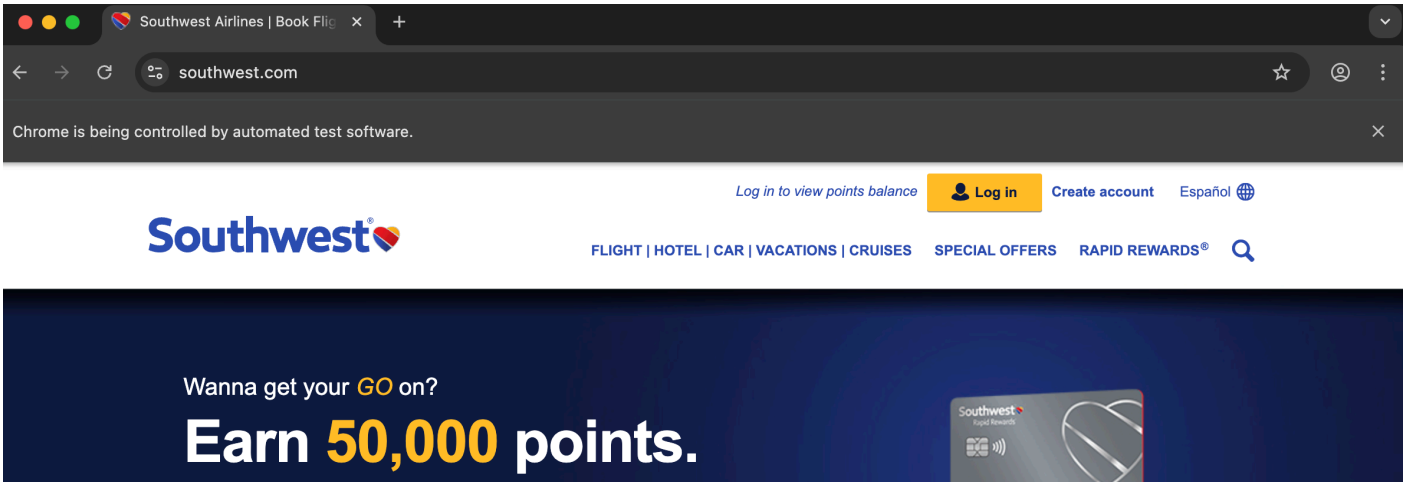
```
from selenium import webdriver

driver = webdriver.Chrome() #creates a chrome driver
```



You should see a empty chrome window pop up when this code is exectued

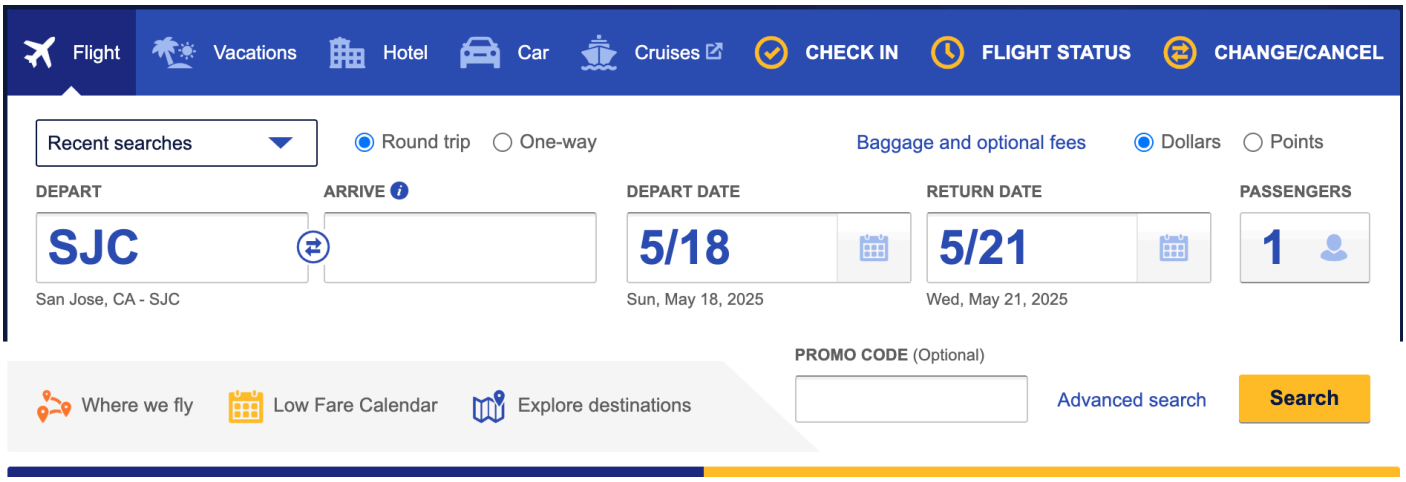
We know we want to go to southwest so let's tell the driver to go there

```
from selenium import webdriver

driver = webdriver.Chrome() #creates a chrome driver
driver.get('https://www.southwest.com')
```



The `get` method tells selenium to go to a certain page.

Manipulating the form



So you could spend your time finding all of the form elements, manipulating them, and then submitting. However I ran into some problems on this domain with this approach. Depending on the screen size the form element ids/classes change. Sometimes it seemed like a roll of the dice if it would be one format or the other even with the same screen size. Selenium offers some ways to interact with elements with `clear()`, `click()`, and `send_keys()` (as well as some other really cool utilities for doing things like this).

However I am a lazy scraper and this felt like too much work. So I decided to take a shortcut. Earlier we saw some crazy link that encoded all of these values and returned an HTML. If you look at View Based from earlier we can see that the link contains all of these parameters.

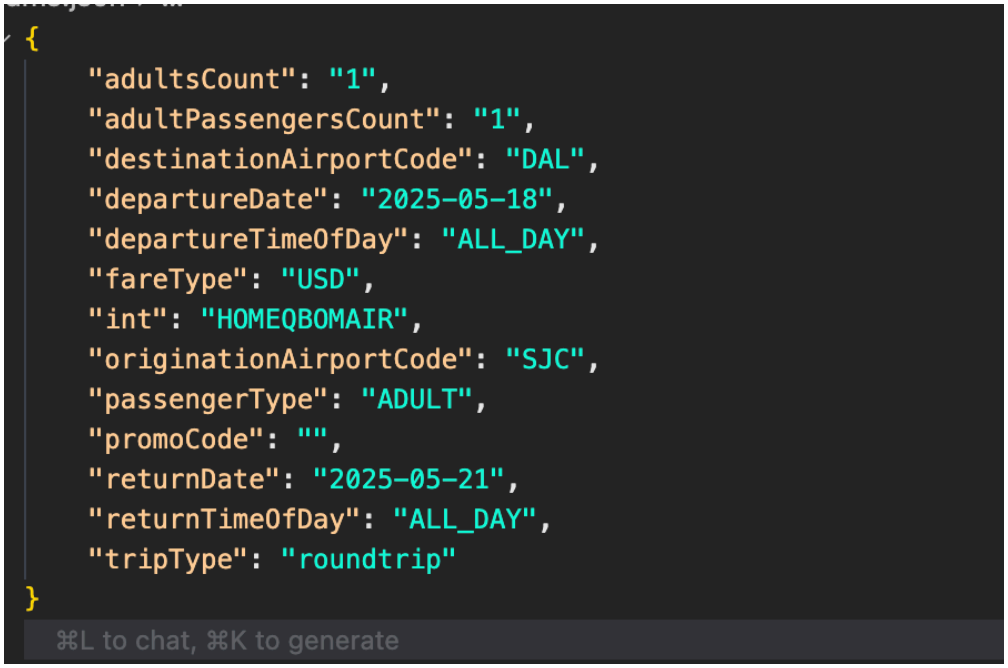
Deconstructing the parameters

Let's parse that link and save it as a json file

`https://www.southwest.com/air/booking/select-depart.html?adultsCount=1&adultPassengersCount=1&destinationAirportCode=DAL&departureDate=2025-05-18&departureTimeOfDay=ALL_DAY&fareType=USD&int=HOMEQBOMAIR&originationAirportCode=SJC&passengerType=ADULT&promoCode=&returnDate=2025-05-21&returnTimeOfDay=ALL_DAY&tripType=roundtrip`

We can see that the main endpoint it separated from the query by the `?` symbol. We can also see that each key, value pair is separated by a `&`, and that each actual key and value is separated by an `=`.

```
import json
url = 'https://www.southwest.com/air/booking/select-depart.html?adultsCount=1&adultPassengersCount=1&destinationAirportCode=DAL&departureDate=2025-05-18&departureTimeOfDay=ALL_DAY&fareType=USD&int=HOMEQBOMAIR&originationAirportCode=SJC&passengerType=ADULT&promoCode=&returnDate=2025-05-21&returnTimeOfDay=ALL_DAY&tripType=roundtrip'

query = url.split('?')[1]
pairs = query.split('&')
params = {item.split("=")[0]:item.split("=")[1] for item in pairs}

with open('params.json', 'w') as f:
    json.dump(params, f, indent=4)
```



We now have a parameter format we can copy and paste wherever we want to and it can serve as a reference for making new parameter options. I'm going to wrap the parameter object inside of a list so that I can iterate through different parameters later. You don't have to do this step but my code will be operating under the assumption that it is receiving a list of parameters so keep that in mind if you decide to diverge from this path.

Reconstructing the URL

Now we want to reconstruct the URL so that we can easily pass the parameters in. We can use pythons built in `urllib` to accomplish this.

```
BASE_URL = 'https://www.southwest.com/air/booking/select-depart.html'

def make_query(args):
    """Create an encoded url using urllib"""
    encoded = urllib.parse.urlencode(args)
    url = f'{BASE_URL}?{encoded}'
    return url
```

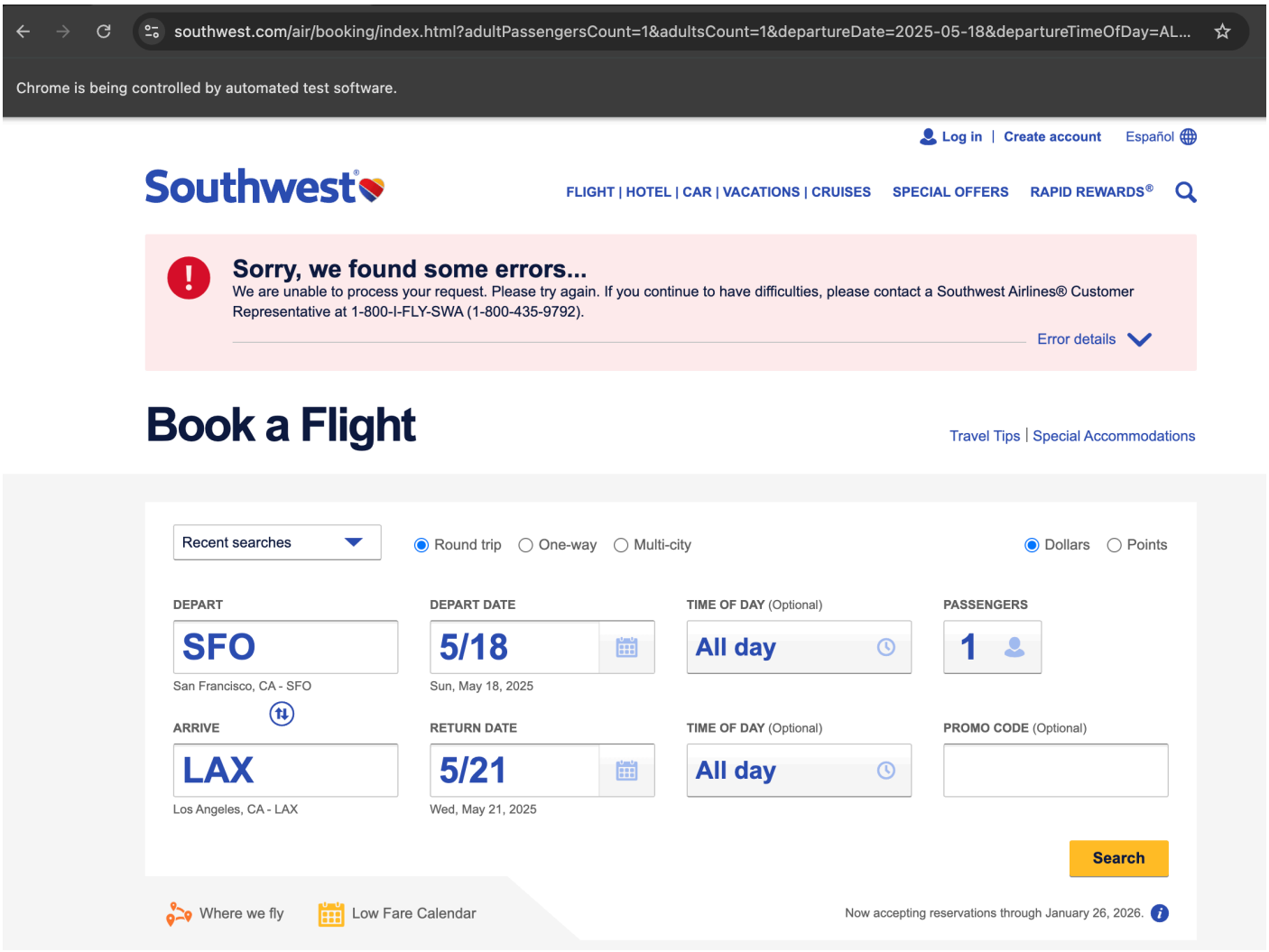
We now have a simple function to go from parameters to URL. Let's sit on this for a second and go back to using selenium.

Skipping the Form

```
import time
from selenium import webdriver
```

```
BASE_URL = 'https://www.southwest.com/air/booking/select-depart.html'

args = { #We have this stored in a file but just for demonstration purposes
right now I will be using this. I changed some of the args for demonstration
purposes
    'adultPassengersCount': '1',
    'adultsCount': '1',
    'departureDate': '2025-05-18',
    'departureTimeOfDay': 'ALL_DAY',
    'destinationAirportCode': 'LAX',
    'fareType': 'USD',
    'int': 'HOMEQBOMAIR',
    'originationAirportCode': 'SFO',
    'passengerType': 'ADULT',
    'promoCode': '',
    'returnDate': '2025-05-21',
    'returnTimeOfDay': 'ALL_DAY',
    'tripType': 'roundtrip'
}

def make_query(args):
    """Create an encoded url using urllib"""
    encoded = urllib.parse.urlencode(args)
    url = f'{BASE_URL}?{encoded}'
    return url

driver = webdriver.Chrome() #creates a chrome driver
driver.get('https://www.southwest.com')

time.sleep(1) # simulate a little bit of waiting

url = make_query(args)
driver.get(url)
```

Running this code you should hopefully see something like this



It says there was an error but don't worry about that right now. We can see that the arguments we used in our mock parameters were placed into the form when we went to the URL.

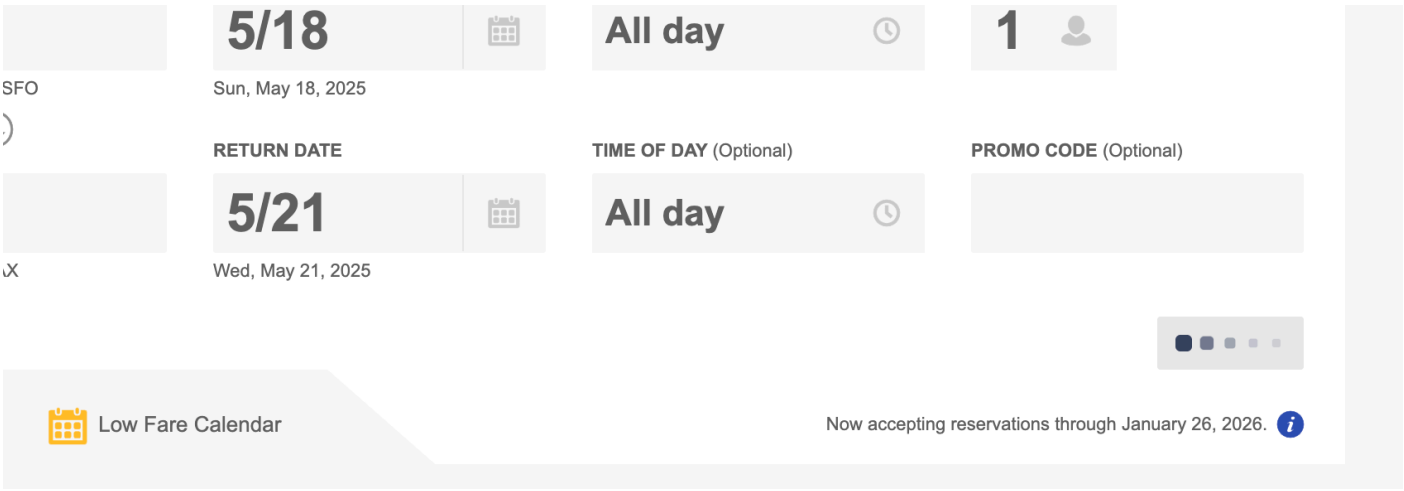
Now what we need to do is click the Search button and we should be good to go.

Getting the Search button

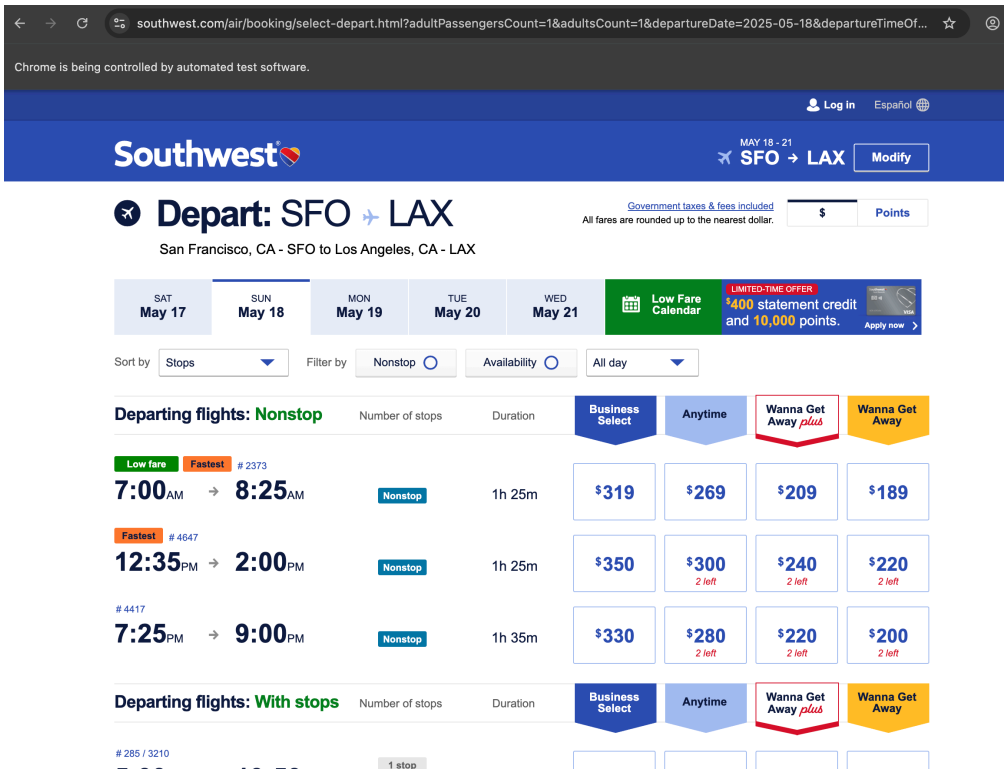
Now we need to utilize the `ImplicitWait` feature of Selenium. An `ImplicitWait` allows our driver to wait until a certain condition is met before performing some action

```
import time
from selenium import webdriver
# additional imports
from selenium.webdriver.common.by import By
from selenium.webdriver.support.wait import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
# ... args and url code from above

driver = webdriver.Chrome() #creates a chrome driver
wait = WebDriverWait(driver, 10) #create a wait object with a 10 second timeout
```

```
on our driver
driver.get('https://www.southwest.com')

time.sleep(1) # simulate a little bit of waiting

url = make_query(args)
driver.get(url)

time.sleep(1) # simulate some more waiting
submit = wait.until(EC.element_to_be_clickable((By.CLASS_NAME,
'actionable_primary'))) #wait til we can click the search button

submit.click() #click the search button
```



Lol this was extremely hard to capture in a screenshot but we can see it was able to click the Search button by the presence of the "..." animation.



Performing this action takes us right to the listing screen of flights (this is what we saw in the View Based section)

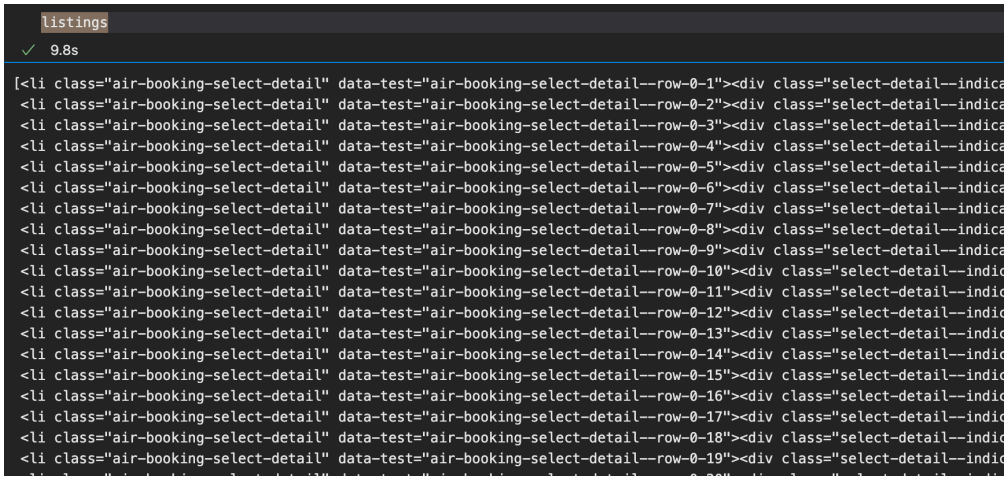
Scraping the HTML in Selenium

Scraping the HTML in Selenium is essentially the same for how we did this in the View Scraping (Naive Approach) section. We just need to get the page source in a different way. We will still be passing it into `beautifulsoup4` and navigating the DOM tree inside of `BeautifulSoup` object.

We are also utilizing the class name for the listings we found in View Based. We are looking for a `li` element with the class `air-booking-select-detail` attached to it

```
# all the code from above taking us to this new page

from bs4 import BeautifulSoup #parse our html dom

shop_items = wait.until(EC.element_to_be_clickable((By.CLASS_NAME, 'air-booking-select-detail'))) #wait for row item to appear

source = driver.page_source
soup = BeautifulSoup(source)

listings = soup.find_all('li', 'air-booking-select-detail') #get our listing elements
```

We are performing another implicit wait to let the listings appear on the page. We then access the `driver.page_source` property to get the html code of the current page we are on in Selenium. Finally we pass this into a `BeautifulSoup` object and we are able to gather listings for the flights!



We finally have data! Now we just need to extract the information we care about.

Data Extraction

This part will involve some digging, and debugging! I have already done this and the code will be available here and in the repo. However you should always practice the digging phase when scraping! Let the inspect tool take you to victory.

```
def extract_listing_info(listing):
    """Extract information from listings"""
    times = listing.find_all('span', 'time--value') #get departure and arrival
times
    real_times = [elem.get_text() for elem in times]

    duration = listing.find('div', 'select-detail--flight-duration').get_text()

    fares = listing.find_all('button', 'fare-button--button')
    fares = [elem.find('span', 'swa-g-screen-reader-only').get_text() for elem
in fares if elem.find('span', 'swa-g-screen-reader-only')]

    stops = listing.find_all('div', 'select-detail--change-planes')
    stops = [stop.get_text() for stop in stops]

    info = {
        'times': real_times,
        'duration': duration,
        'fares': fares,
        'stops': stops
    }
```

```
    return info
```

Putting it all together

I have gone ahead and made some functions to perform this operation and make it usable as a python script. I highly recommend doing your digging and debugging in a notebook file then finally moving it into a callable script once you have worked out most of the kinks.

```
"""Scrape flight information from SouthWest using Selenium"""
import urllib
import time
import json
import argparse

from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.wait import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from bs4 import BeautifulSoup

BASE_URL = 'https://www.southwest.com/air/booking/select-depart.html'

def make_query(args):
    """Create an encoded url using urllib"""
    encoded = urllib.parse.urlencode(args)
    url = f'{BASE_URL}?{encoded}'
    return url

def run_driver(args, sleep_time):
    """Run a Selenium driver and return the page source when we have reached a
state that contains our listings"""
    driver = webdriver.Chrome() #initialize the driver
    wait = WebDriverWait(driver, 10) #set up a wait element (this will timeout
and raise an error if waiting more than 10 seconds)
    query_url = make_query(args)

    driver.get('https://www.southwest.com/') #go to main domain
    time.sleep(sleep_time) #wait a second
    driver.get(query_url) #fire off our url
    time.sleep(sleep_time) #wait again
    submit = wait.until(EC.element_to_be_clickable((By.CLASS_NAME,
'actionable_primary'))) #wait til we can click the search button
    submit.click() #click dat shit
```

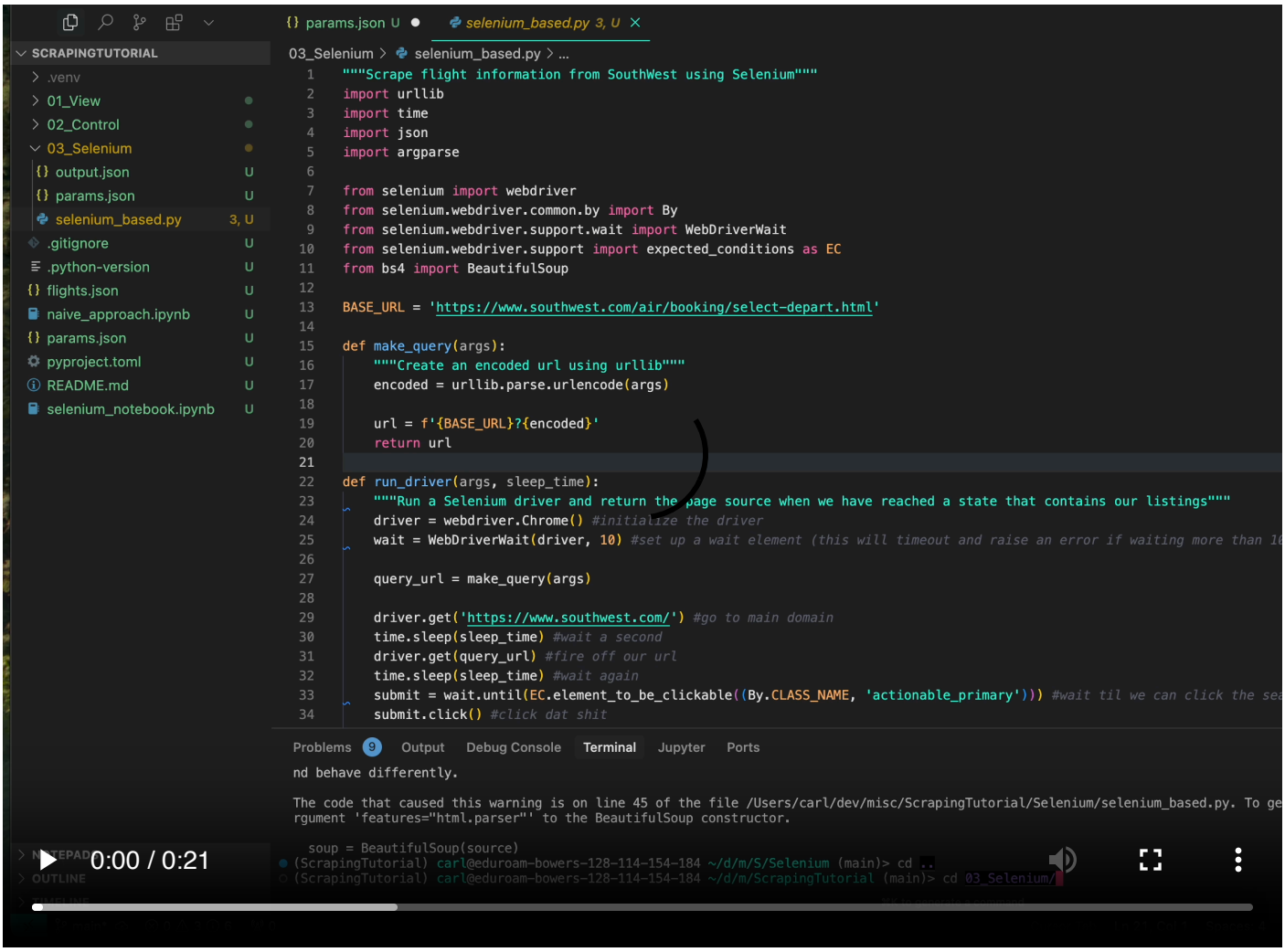
```
    wait.until(EC.element_to_be_clickable((By.CLASS_NAME, 'air-booking-select-
detail'))) #wait for row item to appear

    source = driver.page_source
    driver.close() #close our driver
    return source



def get_listings_from_source(source):
    """Extract our listings from the page"""
    soup = BeautifulSoup(source)
    listings = soup.find_all('li', 'air-booking-select-detail') #get our
listing elements
    assert len(listings) > 0, 'Unable to find listing elements'

    scraped_listings = [extract_listing_info(listing) for listing in listings]

    return scraped_listings




def extract_listing_info(listing):
    """Extract information from listings"""
    times = listing.find_all('span', 'time--value') #get departure and arrival
times
    real_times = [elem.get_text() for elem in times]

    duration = listing.find('div', 'select-detail--flight-duration').get_text()

    fares = listing.find_all('button', 'fare-button--button')
    fares = [elem.find('span', 'swa-g-screen-reader-only').get_text() for elem
in fares if elem.find('span', 'swa-g-screen-reader-only')]

    stops = listing.find_all('div', 'select-detail--change-planes')
    stops = [stop.get_text() for stop in stops]

    info = {
        'times': real_times,
        'duration': duration,
        'fares': fares,
        'stops': stops
    }

    return info
```

```
def main():
"""Main Func"""
parser = argparse.ArgumentParser(
prog='Southwest Ticket Scraper',
description='Gather ticket information from Southwest using selenium and bs4')

parser.add_argument('-p', '--params', default='params.json', help='File for
parameter data')
parser.add_argument('-t', '--time', default=1, help='Time between requests')
parser.add_argument('-o', '--output', default='output.json', help='File to
place output data')

cli_args = parser.parse_args()

with open(cli_args.params, 'r') as f:
    arg_list = json.load(f)

save = []

sleep_time = int(cli_args.time)
for args in arg_list:
    try:
        page_source = run_driver(args, sleep_time)
        listings = get_listings_from_source(page_source)
        output = {
        'params': args,
        'results': listings
        }

        save.append(output)
    except Exception as e:
        print(f'ERROR: Encountered {e}\nUsing {args}')

    time.sleep(sleep_time) #sleep some more

with open(cli_args.output, 'w') as f:
    json.dump(save, f, indent=4)


if __name__ == '__main__':
    main()
```

Here is the script in action

Parameters

DEMO



Now mine actually errored out on the second argument but it did work earlier I swear (lol). Sometimes stuff like this will happen. After erroring out on the second it seemed like southwest rate limited my requests. Sometimes you just have to wait em out.

```
[
    {
        "params": {
            "adultPassengersCount": 1,
            "adultsCount": 1,
            "departureDate": "2025-05-18",
            "departureTimeOfDay": "ALL_DAY",
            "destinationAirportCode": "LAX",
            "fareType": "USD",
            "int": "HOMEQBOMAIR",
            "originationAirportCode": "SFO",
            "passengerType": "ADULT",
            "promoCode": "",
            "returnDate": "2025-05-21",
            "returnTimeOfDay": "ALL_DAY",
            "tripType": "roundtrip"
        },
        "results": [
            {
                "times": [
                    "Departs 7:00AM",
                    "Arrives 8:25AM"
                ],
                "duration": "1h 25m",
                "fares": [
                    "319 Dollars",
                    "269 Dollars",
                    "209 Dollars",
                    "189 Dollars"
                ],
                "stops": []
            },
            {
                "times": [
```


Results from first scrape were saved and we can see that it was able to grab some of the listings.